UNIVERSIDADE FEDERAL DO MARANHÃO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
CURSO DE PÓS-GRADUAÇÃO EM ENGENHARIA DE ELETRICIDADE
ÁREA: CIÊNCIA DA COMPUTAÇÃO

ONTOCADE: UM AMBIENTE *CASE* BASEADO EM ONTOLOGIAS PARA ANÁLISE E PROJETO NA ENGENHARIA DE DOMÍNIO MULTIAGENTE

JOSÉ HENRIQUE ALVES DA SILVA FILHO

São Luís, MA
2005

ONTOCADE: UM AMBIENTE *CASE* BASEADO EM ONTOLOGIAS PARA ANÁLISE E PROJETO NA ENGENHARIA DE DOMÍNIO MULTIAGENTE

José Henrique Alves da Silva Filho
Bacharel em Ciência da Computação
Universidade Federal do Maranhão, 2001

Dissertação apresentada ao Curso de Pós-Graduação em Engenharia de Eletricidade da Universidade Federal do Maranhão como parte dos requisitos para a obtenção do título de Mestre em Engenharia de Eletricidade na área de Ciência da Computação.

Orientadora: Profª. Dra. Rosario Girardi

São Luís, MA
2005

ONTOCADE: UM AMBIENTE *CASE* BASEADO EM ONTOLOGIAS PARA ANÁLISE E PROJETO NA ENGENHARIA DE DOMÍNIO MULTIAGENTE

José Henrique Alves da Silva Filho

Dissertação aprovada em 18/03/2005.

Profª. Dra. Maria del Rosario Girardi
Universidade Federal do Maranhão
(Orientadora)

Prof. Dr. Zair Abdelouahab
Universidade Federal do Maranhão

Profª. Dra. Rossana Maria de Castro Andrade
Universidade Federal do Ceará

A Deus, meus pais e entes queridos.

"Que a inspiração chegue não depende de mim. A única coisa que posso fazer é garantir que ela me encontre trabalhando."

Pablo Picasso

AGRADECIMENTOS

A Deus, por ter me iluminado em mais uma etapa da minha vida.

Aos meus pais, José Henrique e Ana Lucia, pela dedicação e compreensão em todos os momentos.

Às minhas irmãs, Lucia Marília e Ana Cristina, pelo apoio e companheirismo.

À Profª Rosario Girardi, pela segura orientação e pelo incentivo constante.

Aos professores do Mestrado em Ciência da Computação, pelos valiosos ensinamentos.

Aos amigos do Ministério Público do Estado do Maranhão, pelo apoio e pelas concessões.

A todos aqueles que, direta ou indiretamente, contribuíram para a elaboração desta dissertação.

RESUMO

Este trabalho propõe *ONTOCADE* (*Ontology-based Environment for Computer-Aided Domain Engineering*), um ambiente *CASE* baseado em ontologias para análise e projeto na Engenharia de Domínio Multiagente (EDMA).

O sucesso da Engenharia de Domínio Multiagente, processo para criação de abstrações de *software* reutilizáveis para a construção de sistemas multiagentes em um domínio ou área de solução de problemas, depende da disponibilidade de metodologias e ferramentas que ofereçam adequadamente suporte para ontologias, comunicações, mobilidade, autonomia e outros tópicos centrais relacionados aos sistemas baseados em agentes.

ONTOCADE fornece suporte à aplicação das fases da *MADEM*, uma metodologia para EDMA. De acordo com as regras de integração para a constituição de um ambiente, as ferramentas que o compõem utilizam um modelo de dados compartilhado e incorporam o conhecimento de métodos para coordenar e guiar a execução das fases do ciclo de desenvolvimento de *software*. Por esta razão, o ambiente *ONTOCADE* é baseado em ontologias, estruturas particularmente apropriadas para representar conhecimento e abstrações de *software* de alto nível, pois apresentam terminologia clara e não ambígua. *ONTOCADE* utiliza uma ontologia genérica, a *ONTOMADEM*, para guiar o usuário na realização da modelagem. Na definição da *ONTOMADEM* é utilizado o conhecimento da *MADEM* para gerar uma rede semântica com a representação dos conceitos da metodologia. No projeto, a *ONTOMADEM* é construída mapeando a rede semântica a uma ontologia representada por uma hierarquia de classes.

Como ambiente de execução do *ONTOCADE*, foi escolhido o *Protégé*, uma plataforma extensível para a criação de aplicações baseadas em conhecimento. A *ONTOMADEM* é uma extensão à meta-ontologia do *Protégé*. Desta forma, o *ONTOCADE* executa como um *plugin*, ou seja, um componente agregado ao *Protégé*, que se serve de todas as funcionalidades providas pela plataforma e oferece suas funções específicas, tais como modelagem de domínio e modelagem arquitetural.

Palavras-chave: Ambientes *CASE*, Sistemas Multiagentes, Ontologias, Engenharia de Domínio, Análise de Domínio, Projeto de Domínio.

Fonte: Silva Filho, José Henrique A. *ONTOCADE*: Um Ambiente *CASE* baseado em Ontologias para Análise e Projeto na Engenharia de Domínio Multiagente. 2005. 129 p. Dissertação (Mestrado em Engenharia de Eletricidade), Universidade Federal do Maranhão, São Luís.

ABSTRACT

This work proposes ONTOCADE (Ontology-based Environment for Computer-Aided Domain Engineering), an ontology-based CASE environment for analysis and design in Multi-Agent Domain Engineering.

Success in Multi-Agent Domain Engineering, the process of creating reusable software abstractions for the development of a set of multi-agent software applications in a domain or problem-solving area, depends on the availability of appropriate methodologies and tools providing adequate support for ontologies, communications, mobility, and autonomy, among others central topics of agent-based systems.

ONTOCADE supports the application of the phases of MADEM, a methodology for Multi-Agent Domain Engineering. According to the rules of integration for composing an environment, its tools use a shared data model and incorporate the knowledge of methods for coordinating and guiding the execution of the phases of the software development lifecycle. For that, ONTOCADE is based on ontologies, structures particularly appropriate for representing knowledge and high-level software abstractions, because they present clear and unambiguous terminology. ONTOCADE uses a generic ontology, ONTOMADEM, for guiding the user in modeling tasks. To define ONTOMADEM, it was used the MADEM knowledge to generate a semantic network with the representation of the concepts of the methodology. For its design, the ONTOMADEM was generated by mapping the semantic network to an ontology represented by a hierarchy of classes.

To function as a runtime environment, it was chosen Protégé, an extensible platform for creating knowledge-based applications. ONTOMADEM is an extension of the meta-ontology of Protégé. In this way, ONTOCADE executes as a plugin, that is, an aggregated component, which takes advantages of all the functionalities provided by Protégé and offers specifics functions, such as domain and architectural modeling.

Keywords: CASE environments, Multi-Agent Systems, Ontologies, Domain Engineering, Domain Analysis, Domain Design.

Source: Silva Filho, José Henrique A. ONTOCADE: An Ontology-based CASE Environment for Analysis and Design in Multi-Agent Domain Engineering. 129 p. Thesis (Graduation in Electrical Engineering), Universidade Federal do Maranhão, São Luís.

SUMÁRIO

LISTA DE TABELAS ..... **XI**

LISTA DE FIGURAS ..... **XII**

ABREVIATURAS E SÍMBOLOS ..... **XIV**

1 INTRODUÇÃO **16**

1.1 Motivação **16**

1.2 Objetivo do Trabalho **17**

1.3 Organização da Dissertação **17**

2 ENGENHARIA DE DOMÍNIO MULTIAGENTE **19**

2.1 Desenvolvimento de *Software* baseado em Agentes **19**

2.1.1 A abstração agente **19**

2.1.2 Tipos de agentes **20**

2.1.3 Sistemas Multiagentes **22**

2.1.3.1 Mecanismos de Cooperação **23**

2.1.3.2 Mecanismos de Coordenação **25**

2.1.4 Metodologias de Desenvolvimento **25**

2.1.4.1 Técnicas para Análise de Requisitos **25**

2.1.4.2 Técnicas para Projeto Arquitetural e Detalhado **29**

2.2 Engenharia de Domínio **32**

2.2.1 Fases e produtos da Engenharia de Domínio Multiagente **33**

2.3 Ontologias **35**

2.3.1 Editores de Ontologias **37**

2.3.1.1 Protégé **37**

2.3.2 *Newspaper*: um exemplo de ontologia **40**

2.4 Considerações Finais **42**

3 ENGENHARIA DE *SOFTWARE* AUXILIADA POR COMPUTADOR **44**

3.1 Ferramentas *CASE* **44**

3.1.1 Histórico 44

3.1.2 Classificação 45

3.1.3 Benefícios 46

3.2 Integração de ferramentas 46

3.3 Ambientes de Desenvolvimento de *Software* 48

3.3.1 Ambiente de Programação 48

3.3.2 *CASE workbench* 49

3.3.3 Ambiente de Engenharia de *Software* 50

3.4 Ferramentas para o Desenvolvimento de Sistemas Multiagentes 51

3.4.1 *JADE* 51

3.4.2 *MAST* 54

3.4.3 *AgentTool* 56

3.4.4 *JAFMAS* 57

3.4.5 *PTK* e *AgentFactory* 59

3.4.6 Tabela Comparativa 61

3.5 Considerações Finais 62

4 *MADEM*: UMA METODOLOGIA PARA ENGENHARIA DE DOMÍNIO MULTIAGENTE 65

4.1 Análise de Domínio 66

4.2 Projeto de Domínio 70

4.3 *ONTOMADEM*: Conhecimento da metodologia *MADEM* 73

4.3.1 Definição da ontologia 73

4.3.2 Projeto da ontologia 76

4.4 Considerações Finais 77

5 *ONTOCADE*: UM AMBIENTE *CASE* BASEADO EM ONTOLOGIAS PARA ANÁLISE E PROJETO NA ENGENHARIA DE DOMÍNIO MULTIAGENTE 78

5.1 Requisitos 78

5.2 Concepção do ambiente 79

5.3 Automatização das atividades da *MADEM* pelo *ONTOCADE* 81

5.3.1 Esboço do projeto 82

5.3.2 Soluções de projeto 92

5.4 Implementação do *plugin* e da interface 96

5.5 Diretrizes de utilização do *plugin* 98

5.6 Considerações Finais 103

6 CONCLUSÕES 104

6.1 Resultados e Contribuições da Pesquisa 104

6.2 Trabalhos Futuros 105

7 APÊNDICE 107

8 ANEXO 120

BIBLIOGRAFIA 123

LISTA DE TABELAS

Tabela 1 - Tarefas de modelagem de algumas técnicas para a análise de SMAs 27
Tabela 2 - Tarefas de modelagem de algumas técnicas para o projeto de SMAs..... 31
Tabela 3 - Alguns editores de ontologias 37
Tabela 4 - Ontologia *Newspaper* 41
Tabela 5 - Funções e atividades auxiliadas pelas ferramentas *CASE* 46
Tabela 6 - Comparação entre algumas ferramentas para o desenvolvimento de SMAs..... 62
Tabela 7 - Fases, tarefas e produtos da *MADEM* 65
Tabela 8 - Superclasse "Conceitos de Modelagem" e suas subclasses..... 95
Tabela 9 - Comparação entre *ONTOCADE* e outras ferramentas baseadas em agentes..... 102

LISTA DE FIGURAS

Figura 1 - As interações de um agente genérico com seu ambiente 20
Figura 2 - Arquitetura de um agente reativo genérico 21
Figura 3 - As fases da Engenharia de Domínio e da Engenharia de Aplicações 33
Figura 4 - Abstrações de *software* e o processo da EDMA 34
Figura 5 - *Newspaper* transcrita para o *Protégé* 42
Figura 6 - Principais componentes de um *CASE workbench* 49
Figura 7 - Dicionário de dados 50
Figura 8 - Arquitetura do Agente Genérico da ferramenta *JADE* 53
Figura 9 - Especificação de um diagrama de papéis em *AgentTool* 57
Figura 10 - Arquitetura da ferramenta *JAFMAS* 58
Figura 11 - Modelos da *PASSI* automatizados pelo *PTK* 60
Figura 12 - Os insumos e os produtos da metodologia *MADEM* 66
Figura 13 - Modelo de Objetivos definido pela *MADEM* 67
Figura 14 - Modelo de Papéis definido pela *MADEM* 67
Figura 15 - Modelo de Pacote definido pela *MADEM* 69
Figura 16 - Modelo de Agentes definido pela *MADEM* 70
Figura 17 - Modelo de Comportamento do Agente definido pela *MADEM* 72
Figura 18 - Processo de construção da *ONTOMADEM* 73
Figura 19 - Rede Semântica dos Conceitos de Modelagem 74
Figura 20 - Rede Semântica da Tarefa de Modelagem de Domínio 74
Figura 21 - Rede Semântica da Tarefa de Projeto Arquitetural 75
Figura 22 - Rede Semântica da Tarefa de Projeto Detalhado 75
Figura 23 - Hierarquia de meta-classes e a meta-classe “modelagem de domínio” 76
Figura 24 - Arquitetura do *ONTOCADE* 80
Figura 25 - Diagramas de Classes do *ONTOCADE* 86
Figura 26 - Diagrama de Colaboração das ferramentas do *ONTOCADE* 87
Figura 27 - Diagrama de atividades da análise de domínio 87
Figura 28 - Diagrama de atividades do projeto de domínio 88
Figura 29 - Diagrama de seqüência da análise de domínio 89
Figura 30 - Diagrama de seqüência do projeto de domínio 89
Figura 31 - Diagrama de Estados da modelagem de um problema 90

Figura 32 - Diagrama de Estados da modelagem de uma área de conhecimento 91
Figura 33 - Diagrama de Estados da fase de projeto de domínio 91
Figura 34 - Arquitetura detalhada do *ONTOCADE* 93
Figura 35 - Exemplo de comentário editado pelo Editor de Notas 94
Figura 36 - Trecho de um índice criado pelo gerador de relatórios 94
Figura 37 - Esqueleto de um *plugin* do tipo *tab-widget* 97
Figura 38 - Configuração de *plugins* 98
Figura 39 - Interface do *ONTOCADE* 99
Figura 40 - Modelador de Objetivos do *ONTOCADE* 99
Figura 41 - Editor Gráfico do Modelador de Objetivos 100
Figura 42 - Processo de instanciação de conceitos 101
Figura 43 - Modelo de Objetivos do *ONTOCADE* 120
Figura 44 - Modelo de Papéis (fase de Análise de Domínio) 121
Figura 45 - Modelo de Papéis (fase de Projeto de Domínio) 122
Figura 46 - Modelo de Interações 122

ABREVIATURAS E SÍMBOLOS

AD*S*	Ambiente de Desenvolvimento de *Software*
AE*S*	Ambiente de Engenharia de *Software*
AP	Ambiente de Programação
API	*Application Program Interface*
AUML	*Agent-based Unified Modeling Language*
CASE	*Computer-Aided Software Engineering*
CASE-W	*CASE Workbench*
CCA	Canal de Comunicação entre Agentes
CLIPS	*C Language Integrated Production System*
CVD*S*	Ciclo de Vida de Desenvolvimento de *Software*
DD	Dicionário de Dados
DDEMAS	*Domain Design for Multi-Agent Systems*
DER	Diagrama Entidade-Relacionamento
DFD	Diagrama de Fluxo de Dados
EDMA	Engenharia de Domínio Multiagente
E*S*	Engenharia de *Software*
FIPA	*Foundation for Intelligent Physical Agents*
FIPA ACL	*FIPA Agent Communication Language*
Gb	*Gigabyte*
GRAMO	*Generic Requirement Analysis Method based on Ontologies*
GUI	*Graphical User Interface*
HTML	*Hypertext Markup Language*
I-CASE	*Integrated CASE*
IDA	Identificador de Agente
JADE	*Java Agent Development Framework*
JAFMAS	*Java-based Agent Framework for Multi-Agent Systems*
JDBC	*Java Database Connectivity*
JESS	*Java Expert System Shell*
JOE	*Java Ontology Editor*
JVM	*Java Virtual Machine*
LED	Linguagem Específica de Domínio

MaAE	*Multi-Agent Application Engineering*
MADEM	*Multi-Agent Domain Engineering Methodology*
MAD*S*	Metodologia baseada em Agentes para o Desenvolvimento de *Software*
MaSE	*Multi-Agent System Engineering*
MAST	*Multi-Agent Systems Tool*
MAST-ADL	*MAST Agent Description Language*
MESSAGE	*Methodology for Engineering Systems of Software Agents*
MOP	*Market-Oriented Programming*
OIL	*Ontology Inference Layer*
OILEd	*OIL Editor*
ONTOCADE	*Ontology-based Environment for Computer-Aided Domain Engineering*
ONTOMADEM	*MADEM Generic Ontology*
OO	Orientação a Objetos
OWL	*Web Ontology Language*
PASSI	*Process for Agent Societies Specification and Implementation*
PINS	*Protégé Instance*
PONT	*Protégé Ontology*
POO	Programação Orientada a Objetos
PPRJ	*Protégé Project*
PTK	*PASSI Toolkit*
RCI	Repositório Central de Informação
RDF	*Resource Description Framework*
SGA	Sistema Gerenciador de Agentes
SGBD	Sistema Gerenciador de Banco de Dados
SMA	Sistema Multiagente
SODA	*Societies in Open and Distributed Agent Spaces*
STM	Sistema de Transporte de Mensagens
TOD-DSL	*Technique based on Ontologies for the Development of Domain Specific Languages*
UML	*Unified Modeling Language*
XML	*Extensible Markup Language*
YP	*Yellow Pages*

1 INTRODUÇÃO

Este trabalho apresenta o *ONTOCADE* (*Ontology-based Environment for Computer-Aided Domain Engineering*), um ambiente *CASE* baseado em ontologias para análise e projeto na Engenharia de Domínio Multiagente (EDMA).

ONTOCADE é uma proposta inserida no contexto do Projeto *MaAE (Multi-Agent Aplication Engineering)* [29], que aborda a complexidade e a produtividade do *software* por meio da construção de técnicas e ferramentas que promovem a reutilização na Engenharia de Aplicações Multiagente.

1.1 Motivação

A Engenharia de Domínio é uma abordagem sistemática para a construção de abstrações de *software* reutilizáveis [28]. Uma abstração de *software* é uma especificação de alto nível que descreve o que o artefato de *software* faz, eliminando os detalhes irrelevantes e dando ênfase apenas à informação que é importante, nesse nível, para o usuário da abstração.

A abordagem multiagente tem-se apresentado especialmente apropriada para a representação de problemas do mundo real. Por meio da decomposição, um problema complexo é dividido em subproblemas mais simples que podem ser abordados de maneira independente. Na decomposição de *software* baseada em agentes, um problema complexo é dividido em entidades autônomas que interagem de maneira flexível. Este tipo de decomposição reduz o nível de acoplamento entre os componentes do sistema. Como os agentes são ativos e autônomos, eles sabem quando devem atuar e quando devem atualizar seu estado, diferentemente de um objeto passivo, que precisa ser invocado por outro objeto ou entidade externa para atuar. Isso reduz a complexidade de controle que não é mais centralizado, mas localizado em cada agente. Porém, a efetividade do desenvolvimento com reuso dependerá de uma adequada integração das abstrações geradas na Engenharia de Domínio, bem como de mecanismos apropriados para por em prática as especificações, seja através de mecanismos de composição ou geração automática [28] [62].

O sucesso no desenvolvimento de sistemas, no entanto, só é alcançado com o emprego de metodologias e ferramentas *CASE* apropriadas, visando simplificar o trabalho dos

projetistas, automatizar a construção de artefatos de *software* e, conseqüentemente, aumentar a produtividade e reduzir custos [13]. As ferramentas *CASE* estão, cada vez mais, tornando-se cruciais, à medida que a demanda por *software* e a complexidade dos sistemas aumentam. A automatização das fases do ciclo de desenvolvimento possibilitada por estas ferramentas traz inúmeros benefícios, dentre os quais, destacam-se: incremento da qualidade, prototipação e facilitação da manutenção.

A integração de ferramentas é um fator essencial para assegurar a efetividade da tecnologia *CASE*. As ferramentas devem possuir um repositório central para armazenar toda a informação sobre modelos e facilitar a execução de tarefas, tais como controle de consistência, geração de relatórios e reuso de componentes e diagramas [18]. Para que a integração seja bem sucedida, as ferramentas precisam incorporar conhecimento sobre as técnicas assistidas [64]. Desta forma, o uso de ontologias é conveniente, pois estas estruturas representam modelos de conhecimento que podem ser usados para guiar as atividades do processo de desenvolvimento de aplicações.

Vários produtos já foram gerados no Projeto *MaAE* [29], entre técnicas, metodologias, ontologias genéricas e linguagens específicas de domínio (LEDs). Uma das metodologias formuladas é a *MADEM (Multi-Agent Domain Engineering Methodology)* [20], proposta para facilitar a aplicação das fases de análise e projeto da Engenharia de Domínio Multiagente. No entanto, a Engenharia de *Software* (ES) moderna não pode ser realizada sem o suporte inteligente de ferramentas [32]. Por conseguinte, este trabalho propõe a criação de um ambiente *CASE* para automatizar as atividades que compõem as fases da *MADEM* [20].

1.2 Objetivo do Trabalho

Este trabalho visa conceber e desenvolver um ambiente *CASE* baseado em ontologias para fornecer suporte à aplicação de diretrizes metodológicas para análise e projeto na Engenharia de Domínio Multiagente. Serão abordados os requisitos, a arquitetura e o projeto do ambiente, bem como a implementação de sua interface.

1.3 Organização da Dissertação

Esta dissertação é formada por seis capítulos.

O capítulo 2 aborda a Engenharia de Domínio Multiagente e apresenta as definições e metodologias relacionadas ao desenvolvimento de *software* baseado em agentes. Apresenta-se uma visão geral das ontologias, estruturas adequadas para a representação de conhecimento de domínio de uma aplicação em particular.

O capítulo 3 descreve a Engenharia de *Software* Auxiliada por Computador (tecnologia *CASE*). Este capítulo mostra a classificação e os benefícios das ferramentas *CASE*, bem como a integração das mesmas em ambientes de desenvolvimento, fator fundamental para o sucesso da tecnologia.

O quarto capítulo descreve a *MADEM* [20], uma metodologia que visa facilitar a execução das fases de análise e projeto na Engenharia de Domínio Multiagente e, a *ONTOMADEM* [20], uma ontologia genérica que representa o conhecimento da *MADEM*.

O capítulo 5 introduz o *ONTOCADE*, um ambiente *CASE* baseado em ontologias para análise e projeto na Engenharia de Domínio Multiagente. Este ambiente fornece suporte à aplicação da *MADEM*.

O sexto capítulo destaca os resultados obtidos e as contribuições do trabalho, bem como faz sugestões para trabalhos futuros.

2 ENGENHARIA DE DOMÍNIO MULTIAGENTE

A Engenharia de *Software* baseada em agentes fornece soluções para abordar a crescente complexidade dos sistemas de computação, que geralmente devem operar em ambientes imprevisíveis, abertos e que mudam rapidamente. Estes sistemas devem ser capazes de decidir, por si próprios, o que fazer em qualquer situação, para alcançar seus objetivos [28] [29].

O reuso de *software*, fundamental para o sucesso no desenvolvimento de sistemas, é caracterizado por duas abordagens no paradigma de agentes: a Engenharia de Domínio (ou desenvolvimento *para* o reuso) e a Engenharia de Aplicações (ou desenvolvimento *com* o reuso).

A Engenharia de Domínio busca a construção de abstrações de *software* reutilizáveis, que serão usadas na criação de aplicações específicas no desenvolvimento com reuso.

A Seção 2.1 mostra um panorama do desenvolvimento baseado em agentes, destacando os principais conceitos e metodologias. A Seção 2.2 aborda o processo da Engenharia de Domínio Multiagente. Na Seção 2.3 discutem-se as ontologias, estruturas ideais para a representação de conhecimento de domínio e áreas de resolução de problemas.

2.1 Desenvolvimento de *Software* baseado em Agentes

Esta seção aborda o paradigma computacional baseado em agentes, que surgiu como uma evolução da programação orientada a objetos (POO), de forma a atender mais eficientemente os novos domínios de aplicações.

2.1.1 A abstração agente

Um agente [28] é uma entidade autônoma que percebe seu ambiente através de sensores e age sobre o mesmo por intermédio dos executores (Figura 1). O agente possui uma memória interna que será atualizada com a chegada de novas percepções. Essa memória é utilizada nos procedimentos de tomada de decisão, os quais irão gerar ações. De forma a

manter um histórico do comportamento do agente, a memória é também atualizada a partir da ação selecionada.

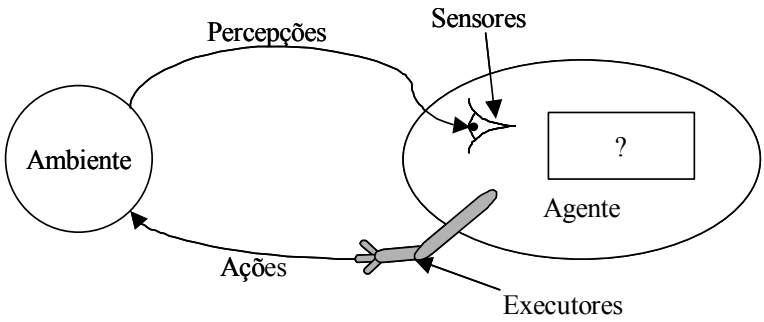

Figura 1 - As interações de um agente genérico com seu ambiente [28]

A autonomia é a característica principal dos agentes. Estes são capazes de atuar sem intervenção humana ou de outros sistemas, por meio do controle que eles possuem do seu estado interno e do seu comportamento. Além de autônomo, o comportamento do agente deve ser flexível o suficiente para que ele alcance seus objetivos. Por esta razão, o agente deve ser reativo, pró-ativo e socialmente hábil:

a) **Reatividade**. Os agentes percebem seu ambiente e respondem, instantaneamente, às mudanças que nele ocorrem para alcançar seus objetivos;

b) **Pró-atividade**. Os agentes são capazes de exibir comportamento orientado a objetivos, tomando a iniciativa de forma a alcançar suas metas;

c) **Habilidade social**. Os agentes são capazes de interagir com outros agentes e, possivelmente, com humanos, para alcançar seus objetivos.

2.1.2 Tipos de agentes

Basicamente, as arquiteturas de agentes diferem uma das outras pelo mecanismo utilizado para decidir a ação a partir de uma percepção. Nesse sentido, as arquiteturas podem ser classificadas em dois tipos fundamentais: deliberativa e reativa [28] [29] [62].

Os agentes deliberativos possuem um modelo simbólico do ambiente. A tomada de decisão da ação a partir de uma percepção é realizada via raciocínio lógico, de acordo com o plano de ação elaborado para atingir a meta.

Os agentes reativos não possuem um modelo do ambiente. Eles agem respondendo a um estímulo do ambiente. O comportamento inteligente pode ser gerado sem representações simbólicas e raciocínio lógico, mas como uma propriedade emergente de certos sistemas complexos, ou seja, a inteligência é resultado da interação do agente com o ambiente e, portanto, os agentes reativos podem exibir também comportamento inteligente. A Figura 2 mostra este tipo de arquitetura.

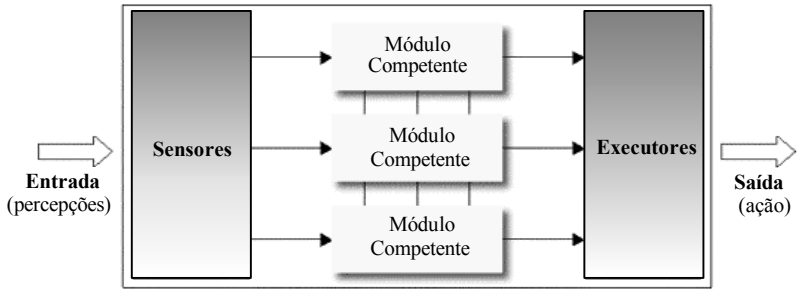

Figura 2 - Arquitetura de um agente reativo genérico [28]

Cada módulo é responsável por uma tarefa definida, não necessariamente complexa, e deve possuir todos os recursos necessários para executá-la. É possível que esse tipo de agente não consiga resolver qualquer tarefa, pois há a possibilidade de não existir um módulo competente para ela. Uma vez que os módulos trabalham em paralelo, deve existir um mecanismo simples para permitir a comunicação entre os módulos.

A estrutura em módulos da arquitetura reativa é descentralizada, o que aumenta a tolerância a erros e a robustez do agente. Ao contrário dos agentes deliberativos, a falha de qualquer módulo não implica na falha do sistema. Os módulos competentes têm seus objetivos implícitos que são definidos por suas funcionalidades e não podem ser modificados. Diferentemente dos agentes deliberativos, eles não podem usar seu conhecimento interno para gerar dinamicamente seus novos objetivos. Apesar desta restrição, muitos pesquisadores concordam que os agentes reativos são capazes de ações orientadas a objetivos. De forma similar à inteligência, a orientação a objetivos é resultado da interação com o ambiente e não de uma avaliação ou planejamento centralizado [28].

2.1.3 Sistemas Multiagentes

Um sistema multiagente (SMA) pode ser caracterizado como um grupo de agentes que atuam em conjunto no sentido de resolver problemas que estão além das suas habilidades individuais [28] [62]. Os agentes interagem entre si de modo cooperativo para atingir uma meta.

A construção de *software* de alta qualidade não é tarefa simples, principalmente devido à complexidade natural do *software*, caracterizada pelas interações necessárias entre seus componentes. Com o objetivo de desenvolver técnicas apropriadas para abordar a complexidade do *software*, os paradigmas de desenvolvimento têm evoluído da programação estruturada à orientação a objetos (OO) e desta, à orientação a agentes.

Pela decomposição, um problema complexo é dividido em subproblemas mais simples que podem ser abordados de maneira independente. Concentrando-se em um determinado momento só em um problema simples, os projetistas podem controlar a complexidade de um problema.

Na decomposição de *software* baseada em agentes, um problema complexo também é dividido em subproblemas mais simples: uma comunidade de agentes autônomos que interagem de maneira flexível.

Um agente é uma abstração de *software* efetiva que fornece uma boa metáfora para a modelagem de sistemas complexos devido, principalmente, à forte correspondência entre a noção de subsistema e de sociedade multiagente. Ambos envolvem um conjunto de componentes agindo e interagindo de acordo com seus papéis.

O paradigma computacional de agentes simplifica o desenvolvimento de sistemas complexos porque os agentes podem decidir sobre o tipo e escopo das suas interações em tempo de execução. Os agentes são capazes de iniciar uma interação e responder a outros agentes de maneira autônoma e flexível.

A autonomia permite o projeto de agentes capazes de atender requisitos não preestabelecidos. Portanto, a abordagem dos SMAs é adequada para a modelagem de sistemas abertos, porque os agentes são capazes de reagir a eventos imprevistos, explorando as oportunidades que surgem e realizando dinamicamente acordos com outros agentes que, porventura, não tiveram suas existências previstas no projeto do SMA.

A arquitetura de um SMA mostra o modo como o sistema está implementado em termos de propriedades e estrutura, e como os agentes que o compõem podem interagir entre si com o objetivo de garantir a funcionalidade do sistema.

De acordo com a sua complexidade, a arquitetura de um sistema multiagente pode ser classificada em três grupos:

a) **Arquitetura simples**. Quando é composta por um único e simples agente;

b) **Arquitetura moderada**. Quando é composta por agentes que realizam as mesmas tarefas, mas possuem diferentes usuários e podem residir em máquinas diferentes;

c) **Arquitetura complexa**. Quando é composta por diferentes tipos de agentes, cada um com certa autonomia, podendo cooperar e estar em diferentes plataformas.

As arquiteturas dos sistemas multiagentes podem ser também classificadas de acordo com os mecanismos de cooperação e coordenação utilizados na sociedade.

2.1.3.1 Mecanismos de Cooperação

Nos sistemas multiagentes, a existência de uma política de cooperação é essencial, uma vez que é por meio deste mecanismo que os agentes exprimem suas necessidades a outros agentes, a fim de atingir determinados objetivos [28].

O processo de cooperação pode ocorrer de duas maneiras: partilha de tarefas e partilha de resultados. A primeira caracteriza-se pela necessidade de agentes auxiliares durante a execução de uma tarefa por um determinado agente, enquanto que na segunda, os agentes compartilham informações com a sociedade, levando-se em conta a possibilidade de que algum outro agente possa precisar delas em determinado momento.

Desta forma, na maioria das vezes, durante a realização de tarefas, os agentes necessitam da ajuda dos outros integrantes da agência ou auxiliam outros agentes para que o objetivo geral do sistema possa ser alcançado. Conseqüentemente, a fácil localização de um determinado agente dentro da sociedade se faz necessária para que a cooperação possa ser efetuada.

As principais abordagens arquiteturais de SMAs, onde a cooperação está em destaque são: a arquitetura quadro-negro, a arquitetura de troca de mensagens e a arquitetura federativa.

Em uma sociedade baseada na *arquitetura quadro-negro*, os agentes não se comunicam entre si de maneira direta, mas sempre através de um quadro-negro, estrutura de dados persistente, onde existe uma divisão em regiões ou níveis visando facilitar a busca de informações. Ele é um meio de interação entre os agentes (uma espécie de repositório), onde estes escrevem e lêem mensagens que serão usadas para atingir o objetivo do sistema. Pode-se afirmar que um quadro-negro é uma memória de compartilhamento global onde existe uma quantidade de informações e conhecimento usados para leitura e escrita pelos agentes. Quando os agentes necessitam de alguma informação, eles escrevem seu pedido no quadro à espera que outros agentes respondam à medida que acessem o mesmo. Este mecanismo, no entanto, é inviável para sistemas de tempo real, pois o processo de gravação e recuperação pode se tornar demorado demais.

Na *arquitetura de troca de mensagens*, os agentes se comunicam diretamente, uns com os outros, por meio de mensagens assíncronas. É necessário, portanto, que cada agente saiba os nomes e endereços de todos os agentes que formam o sistema para que as mensagens possam ser permutadas. Para que as trocas de mensagens ocorram de maneira adequada entre os agentes é preciso o estabelecimento de um protocolo de conversação. O protocolo é quem impõe as regras e o formalismo necessários para que as mensagens sejam transferidas e compreendidas pelos agentes.

Considerando uma arquitetura de troca de mensagens, onde o número de agentes é muito grande, a emissão de uma mensagem em *broadcasting*[1] levará um tempo que pode inviabilizar todo processo de comunicação do sistema. Diante desse contratempo, surgiu a *arquitetura federativa* onde a agência é dividida em grupos ou federações de agentes, conforme um critério de agrupamento escolhido. Junto a cada grupo de agentes encontram-se os agentes chamados de *facilitadores,* responsáveis por receber a mensagem que chega em cada grupo e encaminhá-la para o agente destinatário presente naquele grupo. Este tipo de arquitetura permite a diminuição do fluxo de mensagens desnecessárias entre os agentes que formam a sociedade, pois os facilitadores têm a capacidade de remetê-las ao respectivo destinatário sem a necessidade de enviá-las a todos os agentes.

[1] Envio simultâneo de uma mensagem para múltiplos destinatários.

2.1.3.2 Mecanismos de Coordenação

A coordenação entre agentes refere-se à maneira em que os agentes estão organizados a fim de cooperar para alcançar um objetivo comum do sistema. Existem dois mecanismos básicos para coordenar os agentes: o mecanismo mestre-escravo e o mecanismo de mercado [28].

Nas arquiteturas do tipo *mestre-escravo* há duas classes de agentes: os gerentes (mestres) e os trabalhadores (escravos). Os agentes trabalhadores são coordenados por um gerente que distribui as tarefas e espera o resultado. O agente facilitador pode existir se os agentes estiverem divididos em grupos.

Nas arquiteturas baseadas no *mecanismo de mercado*, todos os agentes encontram-se em um mesmo nível e sabem as tarefas que cada um deles é capaz de executar. Esta arquitetura visa diminuir a quantidade de mensagens trocadas, tendo em vista que cada agente já conhece todos os outros. Baseado nesse mecanismo surgiu a Programação Orientada a Mercado (*MOP*, do inglês *Market-Oriented Programming*), onde os agentes podem ser classificados como produtores e consumidores e são utilizados para maximizar lucros por meio de um processo de negociação [28].

2.1.4 Metodologias de Desenvolvimento

Uma grande quantidade de técnicas e metodologias tem sido proposta para a Engenharia de *Software* baseada em agentes e ainda não existe um padrão reconhecido que reúna o melhor de cada uma delas.

A maioria das propostas aborda as fases de análise de requisitos e projeto de sistemas multiagentes e baseia-se em conceitos das técnicas para o desenvolvimento orientado a objetos ou utilizam conceitos de modelagem da Engenharia do conhecimento [21] [23] [28].

2.1.4.1 Técnicas para Análise de Requisitos

No processo de desenvolvimento de *software*, a fase de análise de requisitos visa definir *o que* o sistema deve fazer e, de acordo com o paradigma de desenvolvimento utilizado, especificar um modelo com a representação dos requisitos do *software* [28].

Os métodos para a análise de requisitos de sistemas multiagentes geralmente enfatizam a modelagem de objetivos, papéis, atividades e interações de indivíduos de uma organização.

Uma organização está composta por indivíduos com objetivos gerais e específicos que estabelecem o que a organização almeja. O alcance de todos os objetivos específicos permite a conquista do objetivo geral da organização. Por exemplo, um sistema de informação pode ter o objetivo geral de "satisfazer as necessidades de informação de uma organização" e os objetivos específicos de "satisfazer as necessidades de informação dinâmica" e de "satisfazer as necessidades de informação em longo prazo".

Os objetivos específicos são alcançados por meio do exercício de responsabilidades que os indivíduos têm. Os indivíduos desempenham papéis com um certo grau de autonomia e exercitam suas responsabilidades pela execução de atividades. Para isso, eles dispõem de recursos. Por exemplo, um indivíduo pode desempenhar o papel de "recuperador de informação" com a responsabilidade de executar atividades para satisfazer as necessidades de informação imediatas de uma organização. Outro indivíduo pode desempenhar o papel de "filtrador de informação" com a responsabilidade de executar atividades para satisfazer as necessidades de informação a longo prazo da organização. Recursos podem ser, por exemplo, as regras da organização para acessar e estruturar suas fontes de informação.

Às vezes, os indivíduos precisam comunicar-se com outros indivíduos, internos ou externos, para cooperar na execução de uma atividade. Por exemplo, os indivíduos que fazem os papéis de "recuperador de informação" e "filtrador de informação" podem precisar interagir com o indivíduo "gerenciador de fontes de informação", que tem a responsabilidade de armazenar e atualizar as fontes de informação da organização.

De acordo com estas explanações, a maioria das técnicas para a análise de requisitos da Engenharia de *software* multiagente estabelece os procedimentos a serem seguidos nas seguintes tarefas de modelagem:

a) **Modelagem de objetivos**. Considerando o problema que o sistema pretende resolver, o objetivo geral do sistema é identificado. Refinando-se objetivo geral, são obtidos os objetivos específicos. Um modelo de objetivos é o produto desta tarefa de modelagem;

b) **Modelagem de papéis**. Para cada objetivo específico, são identificadas as responsabilidades que serão exercidas pelos papéis internos e externos da organização. Então, são definidas as atividades que deverão ser realizadas no exercício de cada responsabilidade. Durante este processo de refinamento, pode ser detectado que a mesma atividade (ou um conjunto de atividades relacionadas) é executada por vários papéis. Neste caso, deve ser criado um papel independente com a responsabilidade de executar estas atividades em nome dos outros papéis. Os recursos que um papel precisará para executar suas atividades são também identificados. Um modelo de papéis é o produto desta tarefa de modelagem;

c) **Modelagem de interações entre papéis**. Analisando-se suas respectivas atividades, são identificadas as interações entre papéis e entre estes e entidades externas. Um modelo de interações é o produto desta tarefa de modelagem.

Além dessas tarefas de modelagem, as versões mais recentes de algumas metodologias propõem também a modelagem dos conceitos do domínio da aplicação de *software* e os relacionamentos entre esses conceitos como base para a construção da(s) ontologia(s) do SMA. Um *modelo de conceitos* é o produto desta tarefa de modelagem.

A Tabela 1 apresenta as tarefas de modelagem de algumas técnicas para a análise de requisitos de sistemas multiagentes: *GAIA* [71], *MaSE (Multi-Agent Software Engineering)* [70], MAD*S* (Metodologia baseada em Agentes para o Desenvolvimento de *Software*) [31], *TROPOS* [6], *SODA (Societies in Open and Distributed Agent Spaces)* [50], *PASSI (Process for Agent Societies Specification and Implementation)* [12] [13], *MESSAGE (Methodology for Engineering Systems of Software Agents)* [5] e COMPOR-M [47].

Tarefas de modelagem	**Técnicas**							
	GAIA	*MaSE*	MAD*S*	*TROPOS*	*SODA*	*PASSI*	*MESSAGE*	COMPOR-M
Modelagem de objetivos		•	•	•	•	•	•	
Modelagem de papéis	•	•	•	•	•	•	•	•
Modelagem de interações	•	•	•	•	•	•	•	•
Modelagem de conceitos		•				•	•	•

Tabela 1 - Tarefas de modelagem de algumas técnicas para a análise de SMAs [28]

Na metodologia *MaSE* [70], por exemplo, os objetivos são mapeados em papéis, mas esse mapeamento é auxiliado pela identificação de casos de uso e pela construção de diagramas de seqüência. A representação dos casos de uso e do diagrama de seqüência é baseada na *UML (Unified Modeling Language)* [4], linguagem para modelagem de sistemas orientados a objetos. Os casos de uso definem cenários básicos de interação com o usuário e são extraídos da especificação de requisitos. Um diagrama de seqüência é construído para cada caso de uso, descrevendo a seqüência de interações entre os papéis do sistema. Além dessas tarefas, *MaSE* também propõe a modelagem do comportamento de cada papel, utilizando a notação dos diagramas de estado da *UML*.

Na metodologia *TROPOS* [6], o conceito de ator substitui o conceito de papel das outras metodologias. Os atores têm objetivos a atingir com a execução de atividades e com o auxílio de recursos. O produto da fase de análise é chamado de *modelo de dependência estratégica*, um grafo onde cada nó representa um ator e os arcos, as dependências entre atores.

A metodologia *PASSI* [12] também utiliza conceitos de modelagem da *UML*. Para a identificação e modelagem de papéis são previamente identificados os casos de uso do sistema, onde os cenários são representados em diagramas de seqüência. Os casos de uso ou pacotes de casos de uso dão origem a agentes. As entidades externas que interagem com os agentes são representadas como atores. Os agentes atuam para alcançar seus objetivos e para isso precisam interagir com outros agentes ou atores. Cada agente pode desempenhar um ou mais papéis, identificados com a construção de diagramas de seqüência que representam as responsabilidades de cada agente em cenários específicos. Um agente pode participar de vários cenários, existindo diferentes papéis em cada um e o mesmo agente pode aparecer mais de uma vez em um diagrama de seqüência específico. Na modelagem de conceitos da *PASSI* são construídos diagramas de classe que descrevem o conhecimento dos agentes, dos atores e de suas interações. A construção do diagrama de classes é baseada na atividade de identificação dos agentes, onde cada agente é introduzido como uma classe no diagrama e, cada mensagem em um diagrama de seqüência é representada como uma associação no diagrama de classes.

A metodologia *MESSAGE* [5] também utiliza conceitos de modelagem da *UML*. *MESSAGE* é baseada nos conceitos de agente, organização, papel, recurso, tarefa, interação e objetivo. Um agente é uma entidade autônoma. Uma organização é um grupo de agentes

trabalhando em função de um objetivo comum. Um papel descreve as características externas de um agente em um contexto particular e um agente pode desempenhar vários papéis. O recurso é usado para representar entidades dependentes, por exemplo, bases de dados. Uma tarefa é uma atividade realizada por um agente. O produto da análise de requisitos é um modelo de visões composto por: *visão da organização*, que representa organizações, agentes, papéis e recursos e seus relacionamentos; *visão das tarefas e objetivos*, representando os objetivos, as tarefas e as dependências entre os mesmos; *visão dos papéis e agentes*, representando os papéis que os agentes desempenham e seus atributos, tais como, objetivo, recursos, tarefas e interações; *visão das interações* e a *visão do domínio*, que representam os conceitos e relacionamentos específicos do domínio.

2.1.4.2 Técnicas para Projeto Arquitetural e Detalhado

No processo de desenvolvimento de *software*, a fase de projeto visa definir uma solução ao problema especificado no modelo de requisitos, de acordo com o paradigma de desenvolvimento utilizado [28]. Os produtos desta fase são uma arquitetura de *software* e uma especificação detalhada dos componentes da arquitetura. Esta fase é geralmente dividida em duas subfases:

a) **Projeto Arquitetural**, onde são definidos os diferentes componentes da arquitetura e a forma em que eles cooperam para a alcançar o objetivo geral do sistema;

b) **Projeto Detalhado**, onde é definido o comportamento e os atributos de dados de cada um dos componentes da arquitetura.

A maioria das técnicas para o projeto na Engenharia de *software* multiagente estabelecem, para a subfase de *projeto arquitetural*, as seguintes tarefas de modelagem:

a) **Modelagem de agentes**. O objetivo desta tarefa é identificar os agentes que irão compor a arquitetura do sistema multiagente e representá-los em um modelo de agentes. Para isso é analisado o modelo de papéis, especificado na fase de análise de requisitos. Os papéis são associados aos agentes numa relação do tipo "*1..n* para *1..n*", de forma a atender requisitos de coesão, desempenho e reusabilidade.

Cada agente irá realizar um conjunto de atividades necessárias para o cumprimento de suas responsabilidades. Estas atividades são derivadas diretamente das atividades dos papéis desempenhados pelos agentes. Vale ressaltar que, nesta tarefa, é realizada apenas a identificação dos agentes que irão constituir a arquitetura do sistema multiagente, sem a preocupação com a arquitetura interna do agente. O produto desta tarefa é um modelo de agentes que especifica os papéis atribuídos a cada agente e um modelo de atividades que especifica as atividades a serem executadas pelo agente.

b) **Modelagem de interações entre agentes**. O objetivo desta tarefa é identificar as interações entre os agentes que compõem a arquitetura do SMA (origem, destino e seqüência) e representá-las em um modelo de agentes. Para isso é realizado um mapeamento das interações entre papéis às interações entre agentes, considerando os papéis que cada agente desempenha. O produto desta tarefa é um modelo de interações que especifica as interações entre os agentes do sistema.

c) **Modelagem da arquitetura do SMA**. O objetivo desta tarefa é especificar, a partir dos agentes e interações identificados nas tarefas anteriores, uma solução computacional ao problema exposto na fase de análise de requisitos. São especificados os mecanismos de cooperação e coordenação entre agentes a fim de que o sistema resolva eficientemente o problema. O produto desta tarefa é um modelo arquitetural do sistema multiagente. Consultando coletâneas ou sistemas de padrões disponíveis, soluções de *projeto arquitetural* podem ser reutilizadas.

Para a subfase de *projeto detalhado*, é estabelecida a tarefa de ***modelagem detalhada dos agentes***. O objetivo desta tarefa é especificar a arquitetura de cada agente da sociedade. A partir de um detalhamento das atividades a serem executadas por um agente, é feita uma análise do seu comportamento e do mecanismo mais apropriado para a tomada de decisão da ação proveniente de uma percepção. Assim, os agentes são estruturados conforme uma estrutura deliberativa ou reativa. O produto desta tarefa é um modelo de *projeto detalhado*, que especifica a arquitetura de cada agente do sistema. Soluções de *projeto detalhado* também podem ser reutilizadas.

A Tabela 2 mostra as tarefas de modelagem de algumas técnicas para o projeto de sistemas multiagentes: *GAIA* [71], *MaSE* [70], MAD*S* [31], *TROPOS* [6], *SODA* [50], *PASSI* [12] [13] e COMPOR-M [47].

Subfases	**Tarefas de modelagem**	**Técnicas**						
		GAIA	*MaSE*	MAD*S*	*TROPOS*	*SODA*	*PASSI*	COMPOR-M
Projeto Arquitetural	Modelagem de agentes	•	•	•	•	•	•	•
	Modelagem de interações	•	•	•	•	•	•	•
	Modelagem da arquitetura do sistema multiagente	•	•	•	•	•	•	
Projeto Detalhado	Modelagem de atividades e dados dos agentes				•	•		•

Tabela 2 - Tarefas de modelagem de algumas técnicas para o projeto de SMAs [28]

Às vezes, as tarefas de modelagem, bem como os produtos construídos, são nomeados de forma diferente da descrição geral anteriormente apresentada. Também, a técnica de construção e representação de um determinado produto pode variar de uma metodologia para outra. Por exemplo, em algumas metodologias, as interações entre agentes são representadas em diagramas de seqüência e, em outras, em diagramas de estado. A seguir são descritas tarefas e produtos de modelagem particulares de cada metodologia, não incluídos na descrição geral de tarefas e produtos feita anteriormente.

Na metodologia *MaSE* [70], além das tarefas especificadas na Tabela 2, é construído um *diagrama de desenvolvimento* que mostra a quantidade e a localização de cada tipo de agente no sistema.

No projeto detalhado dos agentes feito com o auxílio da metodologia *TROPOS* [6], são considerados apenas agentes deliberativos. Portanto, nesta subfase são modelados os eventos internos e externos que disparam planos e crenças envolvidas no raciocínio do agente. As atividades do agente são representadas em um diagrama de habilidades utilizando-se a notação do diagrama de atividades da *AUML (Agent-based Unified Modeling Language)* [44].

Além da fase de projeto, a metodologia *PASSI* [12] define técnicas para a fase de implementação do sistema multiagente. Nesta fase são construídos os modelos de implementação de agentes, de código e de desenvolvimento.

Na metodologia COMPOR-M [47], o comportamento (habilidades) requerido para cada agente é mapeado em serviços providos por componentes funcionais. Caso não existam, serão implementados e estarão disponíveis para reutilização em futuros sistemas. Além da fase de projeto, a metodologia COMPOR-M também fornece suporte às fases de implementação e teste.

2.2 Engenharia de Domínio

A Engenharia de Domínio é uma abordagem sistemática para a construção de abstrações de *software* reutilizáveis [28]. Uma abstração de *software* é uma especificação de alto nível, natural e sucinta, que tem correspondência com uma ou mais realizações num nível de representação mais detalhado. A especificação descreve o que o artefato de *software* faz, eliminando os detalhes irrelevantes e enfatizando a informação que, nesse nível, é importante para o usuário da abstração.

O processo da Engenharia de Domínio é composto pelas fases de análise, projeto e implementação do domínio de uma família de aplicações de *software* [26]. A Figura 3 ilustra as três etapas da Engenharia de Domínio.

As atividades da *Análise de Domínio* identificam oportunidades de reutilização e requisitos comuns de uma família de aplicações no domínio. O produto desta fase é um *Modelo de Domínio*.

No *Projeto de Domínio* busca-se a especificação de uma solução ao problema especificado no modelo de domínio. O produto desta fase é um ou mais *frameworks*[2].

Os componentes reutilizáveis do(s) *frameworks* elaborados na fase anterior são implementados na fase de *Implementação do Domínio*.

Acima foi descrita a abordagem composicional da Engenharia de Domínio. Na sua abordagem gerativa, a Engenharia de Domínio produz LEDs que podem ser usadas como geradores de aplicações para construir uma família de aplicações no domínio [59].

[2] Arquiteturas de *software* reutilizáveis

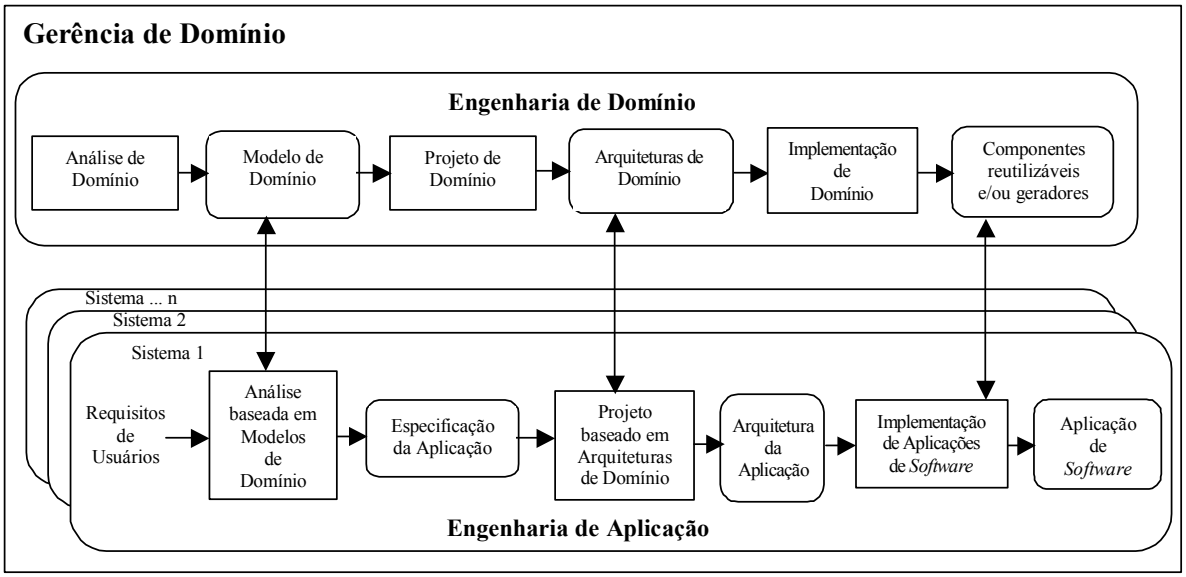

Figura 3 - As fases da Engenharia de Domínio e da Engenharia de Aplicações [26]

2.2.1 Fases e produtos da Engenharia de Domínio Multiagente

Além da complexidade das aplicações, os problemas de produtividade no desenvolvimento, bem como de qualidade e manutenção dos produtos de *software* continuam sendo um desafio para a E*S*. A reutilização de *software* tem apresentado boas soluções para abordar esses problemas [28]. Por isso, a Engenharia de *software* baseada em agentes deve aproveitar os avanços e a experiência adquirida com a teoria e a prática da reutilização de aplicações, em particular, a necessidade de dispor de abstrações de *software* efetivas para a reutilização.

A abordagem multiagente tem-se mostrado particularmente apropriada para representar problemas do mundo real. Porém, a efetividade do desenvolvimento *com* reutilização dependerá de uma adequada integração das abstrações geradas na Engenharia de Domínio, bem como de mecanismos apropriados para mapear especificações em realizações, seja através de mecanismos de composição ou geração automática.

As fases da Engenharia de Domínio Multiagente (Figura 4) são a análise de domínio e usuários, projeto de domínio e implementação de domínio, que produzem abstrações de *software* reutilizáveis, tais como modelos de domínio e usuários, *frameworks* multiagentes, padrões de projeto *global* e *detalhado* e agentes de *software*. As abstrações de *software* reutilizáveis são ilustradas considerando o seu nível de abstração (do abstrato para

concreto) e o seu nível de dependência do domínio da aplicação (do dependente do domínio para o independente do domínio).

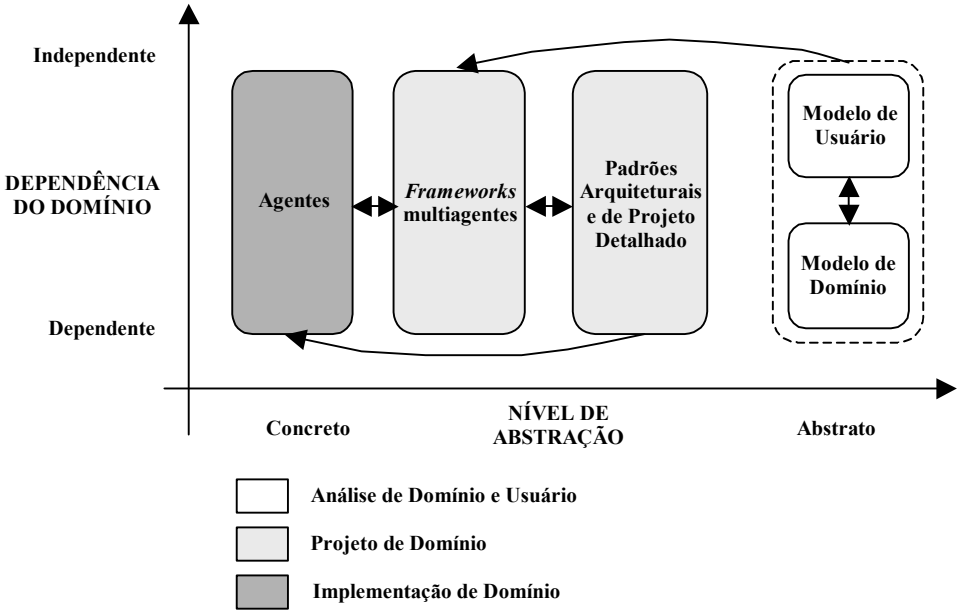

Figura 4 - Abstrações de *software* e o processo da EDMA [29]

A *análise de domínio e usuários* produz uma especificação de requisitos de uma família de sistemas similares em um domínio de aplicação. A especificação inclui a funcionalidade do domínio e as características dos usuários finais. Os modelos de domínio e de usuários são os produtos gerados nesta fase.

O modelo de domínio é uma representação dependente do domínio de aplicação particular, especificada em um alto nível de abstração, que contém a formulação de um problema, conhecimento ou atividade do mundo real. O modelo de usuário, uma abstração igualmente especificada em um alto nível, representa as características, necessidades, preferências e objetivos dos usuários finais.

O *projeto de domínio* produz uma especificação do projeto reutilizável para uma família de sistemas similares em um domínio de aplicação. A especificação do projeto é composta por um conjunto de padrões de projeto *global* e *detalhado* e *frameworks* multiagentes. Os padrões de projeto e os *frameworks* são os produtos gerados nesta fase.

Um *framework* multiagente é uma arquitetura de *software* reutilizável, ou seja, uma solução baseada em agentes para um problema especificado em um determinado

domínio. Ele representa os agentes do sistema e as interações entre eles. Os padrões de projeto são uma solução de *projeto detalhado* a problemas arquiteturais recorrentes.

A *implementação de domínio* transforma as soluções identificadas durante a fase do projeto de domínio em um conjunto de componentes reutilizáveis.

2.3 Ontologias

O conhecimento de domínio é um fator essencial no desenvolvimento de *software*, pois, conhecendo a área de aplicação na qual está trabalhando, o desenvolvedor irá fazer uma boa especificação de requisitos e, conseqüentemente, poderá gerar *software* de boa qualidade [21].

As ontologias são estruturas adequadas para a representação de conhecimento e de abstrações de *software* de alto nível, devido a características como: notação formal, que providencia uma terminologia clara e não ambígua; flexibilidade, que permite posteriores extensões; e a adaptabilidade em diferentes níveis de generalidade/especificidade, facilitando a reutilização.

Uma ontologia [7] é a representação do vocabulário de um domínio. Mais precisamente, não é simplesmente o vocabulário que qualifica a ontologia, mas os conceitos que os termos do vocabulário pretendem capturar.

As ontologias de domínio são descrições formais de classes de conceitos e relacionamentos entre esses conceitos que descrevem um domínio de aplicação. Estruturas de representação de conhecimento baseadas em *frames*[3] são geralmente utilizadas para representar ontologias de domínio, onde os conceitos são representados por *frames* e os relacionamentos entre conceitos como atributos dos *frames*.

As ontologias são particularmente úteis na representação de abstrações de software reutilizáveis de alto nível como modelos de domínio e *frameworks*. Elas fornecem uma terminologia não ambígua que pode ser compartilhada por todos os atores envolvidos em um processo de desenvolvimento. Por outro lado, uma ontologia pode ser generalizada, tanto quanto necessário, facilitando assim sua reutilização e adaptação.

[3] Estrutura de dados ou objeto usado para representar algum conceito em um domínio (ex: classes e instâncias).

Uma razão fundamental para a utilização de ontologias no processo de desenvolvimento de sistemas multiagentes é que elas são necessárias para viabilizar a comunicação entre agentes [7] [28]. Os agentes se comunicam através de mensagens que contém expressões formuladas em termos de uma ontologia. Para que os agentes possam entender o significado dessas expressões, eles precisam acessar uma ontologia comum.

Com a utilização de ontologias, é possível definir uma infra-estrutura para integrar sistemas inteligentes no nível do conhecimento, trazendo as seguintes vantagens [21] [49]:

a) **Representação do conhecimento e reuso**. Uma ontologia representa um vocabulário de consenso e especifica, por meio de uma terminologia clara, o conhecimento de domínio no seu mais alto nível de abstração, com enorme potencial para o reuso;

b) **Comunicação**. O uso de um vocabulário padronizado facilita a troca de informações entre os componentes do sistema;

c) **Formalização**. As ontologias fornecem uma notação formal que elimina possíveis ambigüidades;

d) **Colaboração**. O uso de ontologias permite aos membros da equipe compartilhar o conhecimento;

e) **Interoperação**. Em aplicações distribuídas, a integração de informações é facilitada pelo uso de ontologias;

f) **Informação**. As ontologias podem ser usadas como fontes de pesquisa e de referência do domínio;

g) **Modelagem**. As ontologias podem ser reutilizadas na modelagem de sistemas no nível do conhecimento;

h) **Busca baseada em ontologias**. Com o uso de ontologias, as pesquisas tornam-se mais rápidas, pois a estrutura semântica de uma ontologia permite, não só a entrega de resultados exatos, mas também, respostas próximas à especificação da consulta.

2.3.1 Editores de Ontologias

Ferramentas para o desenvolvimento de ontologias devem fornecer suporte à definição e modificação de conceitos, relações, propriedades, axiomas e restrições, além de possibilitar a inspeção e codificação das ontologias construídas. A Tabela 3 mostra alguns exemplos de editores de ontologias e suas particularidades.

EDITOR	DESCRIÇÃO
Java Ontology Editor (JOE) [37]	Permite a formulação de consultas em vários níveis de abstração e utiliza diagramas de entidade-relacionamento para a representação de ontologias, editadas a partir de uma interface gráfica com o usuário.
OILEd (OIL Editor) [2] [46]	Fornece suporte à construção, integração e verificação de ontologias, escritas na linguagem de ontologias *OIL (Ontology Inference Layer)* [45]. Possibilita a definição de conceitos, relações e alguns axiomas.
OntoEdit [51] [65] [66]	Gera especificações de axiomas de categorização a partir da modelagem de conceitos e relações. A categorização é baseada em significados semânticos ao invés de representações sintáticas.
Ontolingua Server [22]	Permite acesso remoto a ontologias, que podem ser criadas, mantidas e compartilhadas por intermédio de navegadores *web*. Provê acesso a bibliotecas de ontologias.
Protégé [55]	Utilizado para captura de conhecimento. Uma ontologia é representada por classes, instâncias dessas classes, *slots* (atributos de classes e instâncias) e facetas (informações adicionais para os *slots*).

Tabela 3 - Alguns editores de ontologias

2.3.1.1 Protégé

Como ambiente de execução do *ONTOCADE*, foi escolhido o *Protégé* [55]. Desenvolvido no Laboratório de Informática da Escola de Medicina de Stanford (Califórnia, EUA), este editor de ontologias é, também, um editor de bases de conhecimento. *Protégé* [55] é uma plataforma de código aberto, escrita em Java [16] [34], que provê uma arquitetura extensível para a criação de aplicações baseadas em conhecimento. Apesar do desenvolvimento desta ferramenta ter sido, historicamente, ligado a aplicações biomédicas, *Protégé* é independente de domínio e tem sido largamente usado com sucesso em várias outras áreas de aplicações.

Em termos funcionais, *Protégé* é uma:

a) **Ferramenta** que permite ao usuário construir uma ontologia de domínio (definição de classes e hierarquias de classes), gerar formulários padronizados com base nos *slots* especificados e definir instâncias das classes da ontologia;

b) **Plataforma** que pode ser estendida com *widgets*[4] gráficos para tabelas, diagramas, componentes de animação para acessar outras aplicações que englobam sistemas baseados em conhecimento;

c) **Biblioteca** que outras aplicações podem usar para acessar e mostrar bases de conhecimento.

Assim como outras ferramentas de modelagem, a arquitetura do *Protégé* está dividida em duas partes: *modelo* e *visão*. O *modelo* permite a representação interna de ontologias e bases de conhecimento. A *visão* provê uma interface gráfica para exibir e manipular os modelos [36].

Fundamentalmente, o *modelo* é baseado num metamodelo flexível, comparável aos sistemas orientados a objetos e baseados em *frames*, que possibilita a representação de uma ontologia em termos de classes, propriedades (*slots*), características (facetas e restrições) e instâncias. *Protégé* provê uma *API* (*Application Program Interface*)[5] Java para consultar e manipular modelos. É importante destacar que o próprio metamodelo do *Protégé* é uma ontologia, cuja estrutura pode ser facilmente estendida e adaptada a outras representações, como *UML* [4] e *OWL* (*Web Ontology Language*) [36] [52].

A interface gráfica do *Protégé* permite aos projetistas a criação de classes, definição de propriedades, atribuição de facetas e restrições aos *slots*. Utilizando as ontologias construídas, *Protégé* é capaz de gerar automaticamente interfaces de usuários que dão suporte à criação de instâncias. Para cada classe da ontologia, a plataforma cria, em função de cada propriedade da classe, um formulário com *widgets*, que são utilizados para criar e editar as instâncias de classes e ainda, são úteis para visualizar redes de instâncias e relacionamentos entre instâncias. Por exemplo, para propriedades que só podem receber valores do tipo "*string*", o sistema fornece, por padrão, um *widget* do tipo "campo de texto".

[4] Componentes editáveis; termo genérico para itens gráficos como botões, rótulos, campos de texto, painéis,caixa de combinação, etc.
[5] Conjunto de rotinas, protocolos e ferramentas para construção de aplicações.

Os formulários gerados podem ser aperfeiçoados com o editor de formulários do *Protégé*, onde usuários podem selecionar *widgets* alternativos para seus projetos. A biblioteca de *widgets* predeterminados pode ser incrementada com componentes adaptados ou aprimorados, que farão parte do sistema como *plugins* (componentes agregados).

Outro tipo de *plugin* é o painel (*tab*) de interface. Existem *tabs* para os mais diversos fins, tais como edição de classes, propriedades, formulários e instâncias, além de execução de consultas, interfaceamento com repositórios de dados, visualização gráfica de ontologias e gerenciamento de versões de ontologias.

Além destes dois *plugins*, existe ainda o do tipo *backend*, usado para especificar o mecanismo que o *Protégé* irá lançar mão para armazenamento de dados. Atualmente, *Protégé* pode ser usado para carregar, editar e salvar ontologias em vários formatos, incluindo *CLIPS* (*C Language Integrated Production System*) [11], *XML* (*Extensible Markup Language*) [19], *UML* [4], *OWL* [36] [52], esquemas *RDF* (*Resource Description Framework*) [56], arquivos-texto e bancos de dados relacionais compatíveis com *JDBC* (*Java Database Connectivity*) [16], sendo que estes três últimos são padrões da plataforma.

Protégé permite a criação de ontologias que agregam um elevado número de *frames*. O maior sistema já desenvolvido com a ferramenta possui pouco mais que 150.000 *frames*. Sistemas deste tamanho requerem o uso de um banco de dados adicionado ao sistema como um *plugin* do tipo *backend*. Projetos baseados em arquivos (*CLIPS*, *XML*, *RDF*) que possuem mais de 50.000 *frames* tendem a ser impraticáveis por causa da grande quantidade de memória requerida (mais de 1 *Gb*), bem como tempo significativo de carregamento (aproximadamente 1 minuto). O banco de dados evita estas limitações utilizando *caching*[6]. Desta forma, os *frames* requeridos são trazidos para a memória apenas quando necessários e aqueles que não estão em uso, podem ser removidos.

Toda ontologia criada no *Protégé* é salva como um projeto composto por três arquivos: *nome_do_arquivo.pprj* (PRotégé proJect), que representa o projeto; *nome_do_arquivo.pont* (Protégé ONTology), um arquivo que contém informações sobre classes e *slots*; e *nome_do_arquivo.pins* (Protégé INStance), um arquivo que contém informações sobre instâncias. O arquivo *.pprj* é usado para abrir a ontologia.

[6] Memória rápida para armazenagem temporária.

2.3.2 *Newspaper*: um exemplo de ontologia

Esta seção apresenta um exemplo simples e intuitivo de ontologia para a representação de dados de jornais: a ontologia *Newspaper* [55], desenvolvida no *Protégé*. Existe uma variedade de usos possíveis para esta base de conhecimento, cujos elementos principais estão descritos a seguir.

a) Uma lista de artigos publicados, indicando a data e a seção de publicação, etc;
b) Informações sobre seções padrões do jornal (Esporte, Negócios, Política, Saúde, etc);
c) Classificados;
d) Publicidade.

Várias aplicações podem usar as informações presentes nesta ontologia. Por exemplo, é possível a construção de:

a) Um sistema para recuperar, organizar e responder consultas sobre artigos publicados;
b) Um sistema para analisar os rendimentos e preços de anúncios;
c) Um sistema para revisar a organização dos empregados, de forma a assegurar que repórteres sejam distribuídos apropriadamente entre os editores e que todas as seções do jornal tenham um editor responsável.

A Tabela 4 descreve algumas classes que fazem parte desta ontologia. São mostrados os *slots* que se associam diretamente a cada classe; com a indicação das superclasses na terceira coluna, pode-se facilmente deduzir quais *slots* são herdados.

De acordo com a documentação da ontologia *Newspaper*, autor é toda pessoa ou organização que assina artigos; editor é a pessoa responsável pelo conteúdo das seções e um repórter é um empregado que escreve matérias. A classe *Conteúdo* é uma superclasse abstrata para as subclasses *Anúncio* e *Artigo*, sobre as quais guarda informações. O vendedor negocia o espaço para publicidade e é responsável pelo conteúdo dos anúncios.

Classe	**Classe equivalente no *Protégé***	**Superclasses correspondentes**	***Slots* diretos**	**Tipo de *Slot***
Administrador	*Manager*	Empregado	-	-
Agência de notícia	*News_Service*	Autor	contato	instância
Anúncio	*Advertisement*	Conteúdo	comprador e vendedor	instância
			nome	*string*
Anúncio padrão	*Standard_Ad*	Anúncio	texto	*string*
Anúncio pessoal	*Personals_Ad*	Anúncio	imagem	*string*
Artigo	*Article*	Conteúdo	nível_leitura e tipo	símbolo
			palavras-chave, cabeçalho e texto	*string*
			autor	instância
Autor	*Author*	*:THING*[7]	nome	*string*
Biblioteca	*Library*	*:THING*	protótipos, anúncios, artigos e organização	instância
Colunista	*Columnist*	Autor e Empregado	-	-
Conteúdo	*Content*	*:THING*	leiaute, seção e publicado_em	instância
			data_expiração	*string*
			número_página	inteiro
			urgente	*boolean*
Diretor	*Director*	Administrador	-	-
Editor	*Editor*	Autor e Empregado	seções e responsável_por	instância
Empregado	*Employee*	Pessoa	salário	*float*
			data_admissão e cargo	*string*
Leiaute	*Layout_info*	*:THING*	-	-
Jornal	*Newspaper*	*:THING*	conteúdos e protótipos	instância
			data	*string*
			número_de_páginas	inteiro
Organização	*Organization*	*:THING*	nome	*string*
			empregados	instância
Pessoa	*Person*	*:THING*	nome, outra_informação e número_fone	*string*
Repórter	*Reporter*	Autor e Empregado	-	-
Vendedor	*Salesperson*	Empregado	-	-

Tabela 4 - Ontologia *Newspaper*

[7] Classe que representa o nó raiz de toda hierarquia de *frames* feita no *Protégé* [55]; mais detalhes no capítulo 5.

A Figura 5 mostra alguns trechos desta ontologia no *Protégé*:

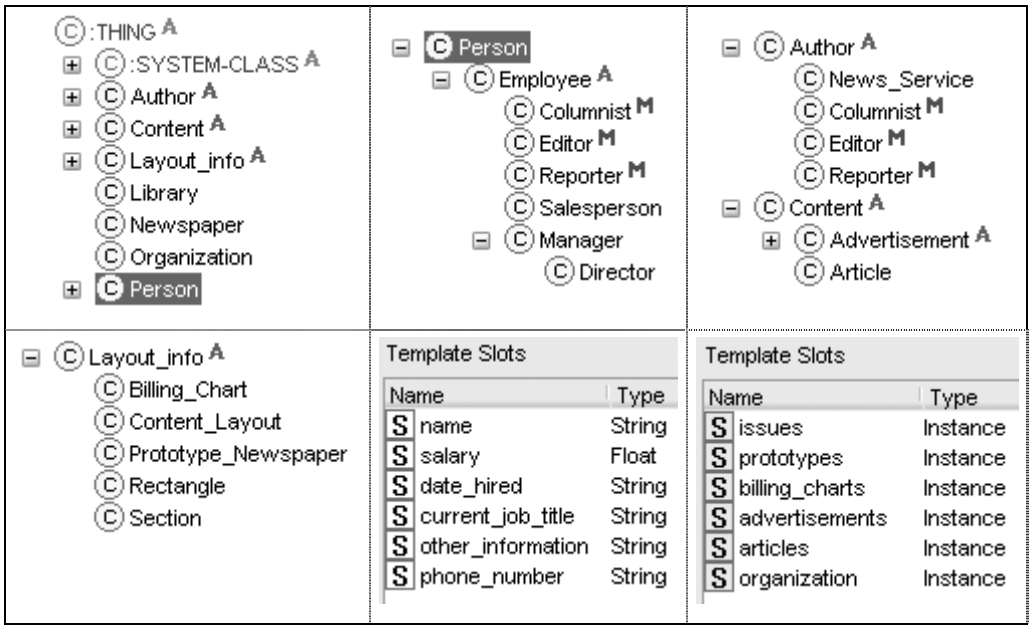

Figura 5 - *Newspaper* transcrita para o *Protégé*

2.4 Considerações Finais

Este capítulo descreveu os principais conceitos relacionados à Engenharia de Domínio Multiagente, uma abordagem sistemática para a construção de abstrações de *software* reutilizáveis. A importância do reuso de *software* foi destacada, uma vez que a reutilização de componentes previamente desenvolvidos, permite a geração rápida de aplicações de qualidade por um baixo custo.

Destacou-se ainda a relevância do paradigma computacional de agentes para abordar a complexidade do *software*, por meio da decomposição de um problema em subproblemas mais simples. Esta é a característica fundamental de um sistema multiagente, formado por entidades (agentes) autônomas, pró-ativas, socialmente hábeis e que interagem entre si para atingir uma meta. Mostrou-se um resumo dos principais tipos de agentes e de algumas metodologias para o desenvolvimento de sistemas multiagentes.

No tocante a Engenharia de Domínio Multiagente, esta é composta por três fases distintas: análise de domínio e usuários, projeto de domínio e implementação de domínio. O objetivo final da EDMA é construir um *framework* multiagente, ou seja, uma solução baseada em agentes para um problema especificado em um determinado domínio.

Assinalou-se que as ontologias são estruturas de conhecimento adequadas para representar abstrações de *software* de alto nível e viabilizar a comunicação entre agentes, pois fornecem uma terminologia não ambígua que pode ser compartilhada por todos os agentes de um sistema. Vários exemplos de editores de ontologias foram citados, entre eles o *Protégé*, plataforma de código aberto que servirá como ambiente de execução para o *ONTOCADE*.

O próximo capítulo abordará a tecnologia *CASE*, mostrando a classificação e os principais benefícios das ferramentas que auxiliam o desenvolvimento de *software*. Será mostrado também como estas ferramentas podem ser integradas para compor verdadeiros *ambientes de desenvolvimento de software* (AD*S*s).

3 ENGENHARIA DE *SOFTWARE* AUXILIADA POR COMPUTADOR

Este capítulo faz uma síntese dos termos e conceitos relacionados à Engenharia de *Software* Auxiliada por Computador, do inglês *Computer-Aided Software Engineering – CASE*.

Ferramentas *CASE* incorporam conhecimento acerca de metodologias e processos da E*S*, fornecendo suporte a cada fase do ciclo de vida de desenvolvimento de *software* (CVD*S*) [41].

A Seção 3.1 apresenta um breve histórico das ferramentas *CASE* e destaca os benefícios de sua utilização. A Seção 3.2 mostra a integração de ferramentas como fator essencial para o sucesso desta tecnologia. Na Seção 3.3 são mostrados os principais tipos de ambientes de desenvolvimento de *software*. A Seção 3.4 apresenta algumas ferramentas para o desenvolvimento de sistemas multiagentes.

3.1 Ferramentas *CASE*

As ferramentas *CASE* automatizam as atividades que compõem o processo de desenvolvimento de *software*. Os principais benefícios oferecidos por estas ferramentas são: flexibilidade, qualidade, redução de custos e facilidade de manutenção [53].

3.1.1 Histórico

A princípio, as ferramentas *CASE* priorizavam a documentação e a diagramação dos sistemas. Editores de texto tornaram-se ótimas ferramentas para documentar as especificações de análise e de projeto. Mais tarde, técnicas gráficas exigiram ferramentas para desenhar diagramas, tais como DFD (Diagrama de Fluxo de Dados) e DER (Diagrama Entidade-Relacionamento) [41][53].

Na geração seguinte, o enfoque passou a ser na automação e controle da análise e projeto, baseados na modelagem de dados e na engenharia de informação. Surgiu o conceito de dicionário de dados (DD), que representa o repositório automatizado de um sistema de

informação. A existência de uma base de informações consistentes acerca do sistema facilita a atualização e a busca de itens da aplicação [41][53][68].

A terceira geração *CASE*, unindo a abordagem gráfica com o dicionário de dados, lançou poderosas ferramentas de desenvolvimento que permitem a derivação de esqueletos de códigos primitivos a partir da especificação de projeto [41].

Com a crescente reutilização, as ferramentas *CASE* passaram a ser usadas na melhoria da interface com o usuário, na prototipação e na definição de padrões de projeto. A quarta geração é formada por ferramentas que, de forma inteligente, guiam a execução de metodologias que automatizam as diferentes atividades do processo de desenvolvimento de *software* [41][53].

Atualmente, existem ferramentas mais especializadas, como aquelas que dão suporte ao reuso de *software* e à reengenharia, e as que utilizam bases de conhecimento, tecnologia empregada em sistemas especialistas [64][68].

3.1.2 Classificação

As ferramentas *CASE* podem ser classificadas segundo duas dimensões: tipo de função provida e tipo de atividade assistida [64].

As ferramentas *orientadas a função* são baseadas nas funções providas, como modelagem, simulação, planejamento, prototipação, processamento de linguagens, dentre outras.

As ferramentas *orientadas a atividade* são baseadas na atividade do processo de desenvolvimento que é auxiliada. Portanto, as ferramentas desta categoria dão suporte às fases de análise, projeto, implementação e verificação.

A Tabela 5 mostra exemplos de funções providas por algumas ferramentas *CASE* e assinala as fases do ciclo de desenvolvimento que recebem o suporte de tais ferramentas. Observa-se que algumas ferramentas estão presentes em todo o ciclo.

FUNÇÕES	FASES DO CICLO DE DESENVOLVIMENTO			
	Análise	Projeto	Implementação	Verificação
Geração de código				•
Modelagem e simulação	•			•
Programação			•	
Depuração			•	•
Processamento de linguagem			•	•
Suporte a métodos	•	•		
Gerenciamento de interface de usuário		•	•	
Gerenciamento de dicionário de dados	•	•		
Diagramação	•	•		
Prototipação	•			•
Gerenciamento de configuração	•	•	•	•
Documentação	•	•	•	•
Edição de texto	•	•	•	•
Planejamento e estimativa	•	•	•	•
Teste				•

Tabela 5 - Funções e atividades auxiliadas pelas ferramentas *CASE* [64]

3.1.3 Benefícios

Os principais benefícios oferecidos pelas ferramentas *CASE* são [68]:

a) Tornam mais prático o uso de técnicas estruturadas, orientadas a objeto ou orientadas a agentes;

b) Provêem qualidade de *software* por meio de revisão automatizada;

c) Simplificam a manutenção de programas;

d) Aceleram o processo de desenvolvimento;

e) Encorajam o desenvolvimento evolucional e incremental;

f) Permitem o reuso de componentes de *software*.

3.2 Integração de Ferramentas

A base para a integração de ferramentas *CASE* é a *enciclopédia*, um repositório central que armazena, de forma consistente e acessível, toda informação referente ao desenvolvimento de um sistema [53][64][68].

As funções principais de uma enciclopédia, salvo algumas peculiaridades, são as mesmas de um sistema gerenciador de banco de dados (SGBD) [61], como impor integridade de dados, compartilhar informações e padronizar o modo de armazenamento dos itens do sistema [53].

Existem três tipos principais de integração [64]:

a) **Integração por dados**. As ferramentas usam um modelo de dados compartilhado. A enciclopédia é a forma mais flexível deste tipo de integração, que pode, ainda, girar em torno de notações de linguagens de programação ou de seqüências não-estruturadas de caracteres, usadas em sistemas operacionais;

b) **Integração por interface**. As ferramentas são integradas em torno de uma interface comum, facilitando, desta forma, tanto a interação usuário-computador quanto a introdução de novas ferramentas no ambiente *CASE*;

c) **Integração por atividade**. O ambiente incorpora um modelo do CVD*S* para coordenar a utilização das ferramentas.

Seja qual for o tipo de integração adotada, os benefícios do *CASE* integrado (*I-CASE*, do inglês *Integrated CASE*), além daqueles já listados na Seção 3.1.3 para ferramentas isoladas, são [38][53]:

a) Envio ordenado de informações de uma ferramenta para outra e de uma atividade da engenharia de *software* para a seguinte;

b) Uso dos recursos de sistemas operacionais e de SGBDs;

c) Suporte às modelagens de dados e de processos, abrangendo, desta forma, não só a engenharia de *software*, mas também a engenharia da informação;

d) Normalização de modelos e diagramas;

e) Geração automática da documentação;

f) Redução do esforço necessário para o gerenciamento de configuração de *software*;

g) Melhorias no planejamento, na monitoração e na comunicação;

h) Coordenação melhorada entre os integrantes da equipe de projetistas.

3.3 Ambientes de Desenvolvimento de *Software*

A integração pode ser parcial, que reúne ferramentas que dão suporte a parte do ciclo de desenvolvimento, ou completa, que permite o auxílio a todo o processo de desenvolvimento de *software*, da análise à verificação.

Um AD*S* é uma coletânea de ferramentas de *hardware* e *software*, utilizada no suporte à construção de sistemas em um domínio de aplicação particular [64]. Portanto, os AD*S*s são os instrumentos mais apropriados para auxiliar o desenvolvimento de sistemas de categorias específicas, como orientação a objeto, orientação a agente, engenharia de conhecimento, e outras da área de inteligência artificial.

Há três grupos principais de AD*S*: Ambiente de Programação (AP), *CASE workbench (CASE-W)* e Ambiente de Engenharia de *Software* (AE*S*) [64]. Apesar da conveniência desta classificação, os limites entre as diferentes classes não são tão claros, uma vez que os avanços experimentados por um grupo tendem a apresentar algumas facilidades inerentes aos demais.

As subseções seguintes tratam das características de cada um destes ambientes integrados de desenvolvimento de aplicações.

3.3.1 Ambiente de Programação

Este foi o primeiro AD*S* a surgir, sendo composto por ferramentas de processamento de linguagem, simuladores, compiladores e ferramentas de teste e depuração. Devido ao seu pioneirismo, os APs possuem ferramentas confiáveis e robustas [64].

Apesar de os APs oferecerem algum suporte para especificação, projeto e gerenciamento, a grande diferença entre estes ambientes e os AE*S*s, está no grau de integração. Como será visto na Seção 3.3.3, enquanto as ferramentas de um AE*S* estão integradas em torno da enciclopédia, as ferramentas em um AP são integradas em torno de um simples arquivo de caracteres seqüenciais [64].

3.3.2 *CASE workbench*

Este ambiente é projetado para dar suporte às fases iniciais do processo de desenvolvimento de *software*: análise e projeto. Além disso, dá suporte a um método específico ou a um conjunto de métodos [64].

Os *CASE workbenches* também são chamados de ferramentas *CASE* superiores (*upper CASE*), pois auxiliam o projetista na produção de especificações completas e consistentes, principais produtos do início do ciclo de desenvolvimento. Em contrapartida, uma ferramenta *CASE* inferior (*lower CASE*) é aquela que dá suporte a atividades como implementação e manutenção [41].

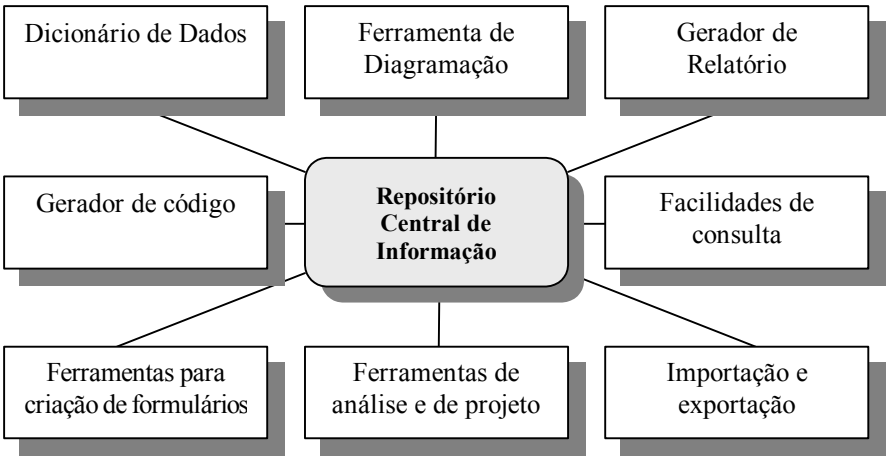

Figura 6 - Principais componentes de um *CASE workbench* [64]

Conforme a Figura 6, os principais componentes de um *CASE-W* são [64][68]:

a) **Editor de diagramas**. Ajuda o engenheiro de *software* a ter uma visão geral do escopo do problema e do fluxo de informação. O editor captura informação sobre as entidades do sistema e armazena-a em um repositório apropriado;

b) **Gerador de relatórios**. Usado para documentar os modelos gráficos e gerar relatórios dos itens de análise e projeto do sistema;

c) **Repositório Central de Informação (RCI)**. É o coração do *CASE workbench*, responsável pelo armazenamento e organização de toda informação acerca de análise e projeto de sistemas. Também chamado de enciclopédia, este componente

é um "conjunto de mecanismos e estruturas de dados que realizam a integração dados-ferramentas e dados-dados" [53]. Assim, o RCI armazena e controla, direta ou indiretamente, o problema a ser resolvido, o conhecimento do domínio, o conhecimento de metodologia e a solução do problema;

d) **Dicionário de dados**. Mantém informação sobre as entidades usadas no sistema. O DD é uma espécie de gramática para descrever o conteúdo de objetos definidos durante a análise de requisitos [53]. A Figura 7 mostra as informações comumente encontradas em um dicionário.

NOME O nome principal de um item de ES (garante a consistência de dados)
ALIAS Nome alternativo para o principal (também garante a consistência)
CLIENTES Lista dos processos que usam o item
MODO DE USO Maneira como o item é usado
COMPLEMENTO Lista de restrições e valores iniciais

Figura 7 - Dicionário de dados

e) **Outras ferramentas**. Compõem, ainda, os *CASE-W*s, as bibliotecas para o reuso, os modeladores, os editores de texto, as ferramentas de prototipação, as ferramentas para a criação de formulários, dentre outras.

3.3.3 Ambiente de Engenharia de Software

Um AE*S*, ou Ambiente *I-CASE* completo, é uma coleção de ferramentas de *hardware* e *software*, que devem, obrigatoriamente, oferecer suporte a todas atividades do processo de desenvolvimento de aplicações, da análise à verificação [38][64].

Os custos iniciais de um AE*S* são maiores que os custos advindos da introdução dos demais AD*S*s. Ambientes de programação e *CASE workbenches* são mais largamente usados, ficando os AE*S*s restritos ao uso em áreas especialistas [64].

3.4 Ferramentas para o Desenvolvimento de Sistemas Multiagentes

Existem algumas ferramentas interessantes para o desenvolvimento de sistemas multiagentes, como *JADE* (*Java Agent DEvelopment Framework*) [33], *MAST* (*Multi-Agent Systems Tool*) [39], *AgentTool* [17], *JAFMAS* (*Java-based Agent Framework for Multi-Agent Systems*) [8], *PTK* (*PASSI Toolkit*) [10] [13], *AgentFactory* [10] [13].

3.4.1 *JADE*

JADE [33] é um *framework* completamente implementado na linguagem Java [34]. Esta ferramenta segue as especificações da *FIPA* (*Foundation for Intelligent Physical Agents*) [24] para sistemas multiagentes. *FIPA* é uma organização cujo objetivo é promover o sucesso de aplicações, serviços e equipamentos baseados em agentes, por meio da colaboração de seus membros (companhias e universidades) nas mais diferentes áreas de aplicações. *JADE* implementa todas as especificações *FIPA* que normatizam o *framework* no qual os agentes podem existir, operar e comunicar.

O objetivo de *JADE* é simplificar o desenvolvimento de sistemas multiagentes, garantindo serviços inerentes ao paradigma de agentes, tais como serviço de nomeação, páginas amarelas, transporte de mensagens e protocolos de interação. A plataforma *JADE* inclui todos os componentes obrigatórios de gerenciamento:

a) **Sistema Gerenciador de Agentes (SGA)**. É o componente que controla o acesso e o uso da Plataforma de Agente. Somente um SGA pode existir em uma plataforma simples. O SGA provê o serviço de páginas brancas (servidor de nomes) e de ciclo de vida, mantendo um diretório de *identificadores de agente* (IDAs) e o estado do agente. Cada agente deve ser registrado no SGA a fim de receber um IDA válido;

b) **Facilitador**. É o agente que provê o serviço de páginas amarelas (oferta de serviços especializados) na plataforma;

c) **Sistema de Transporte de Mensagens (STM)**. Também chamado de *Canal de Comunicação entre Agentes* (CCA), é o componente de *software* que controla

toda a troca de mensagem dentro da plataforma, incluindo as mensagens de/para plataformas remotas.

JADE segue completamente esta arquitetura e quando o ambiente é iniciado, o SGA e o Facilitador são imediatamente criados e o CCA é configurado para permitir a troca de mensagens.

A comunicação entre agentes é feita pela passagem de mensagens, sendo que a linguagem usada para a representação das mesmas é a *FIPA ACL* (*FIPA Agent Communication Language*) [25].

A plataforma de agentes pode ser distribuída por vários *hosts*[8], no entanto, uma única aplicação Java e, portanto, uma única máquina virtual Java (do inglês *Java Virtual Machine*, *JVM*), é executada em cada *host*. Cada *JVM* é, basicamente, um mecanismo que provê um completo ambiente para execução de agentes e permite que vários agentes executem concorrentemente no mesmo *host*.

JADE provê uma interface gráfica de usuário (*Graphical User Interface*, *GUI*) para gerenciamento remoto, monitoração e controle do *status* dos agentes, permitindo, por exemplo, parar e reiniciar agentes ou criá-los em um *host* remoto.

JADE oferece um número de ferramentas gráficas que dão suporte à depuração:

a) ***Dummy***[9]. Inspeciona a troca de mensagens entre agentes e testa a interface de um agente antes de integrá-lo ao SMA. Esta ferramenta provê suporte para edição, composição e envio de mensagens de/para agentes;

b) ***Sniffer***. Permite visualizar todas as mensagens trocadas por um determinado agente;

c) ***Instrospector***. Monitora e controla o ciclo de vida de um agente em execução, bem como das mensagens trocadas, tanto as enviadas como as recebidas.

O agente genérico do ambiente *JADE* é uma arquitetura reativa. Conforme mostra a Figura 8, o modelo computacional do agente genérico *JADE* é multitarefa, onde tarefas (ou comportamentos) são executadas concorrentemente. Cada funcionalidade (ou serviço) provida

[8] Máquina hospedeira; computador que fornece serviços aos demais do sistema ou rede (servidor).
[9] Na falta de termos significativos em português, foram mantidos os originais.

por um agente pode ser implementada com a integração de mais de um comportamento. A definição da ordem de execução das tarefas é feita pelo Gerenciador de Tarefas.

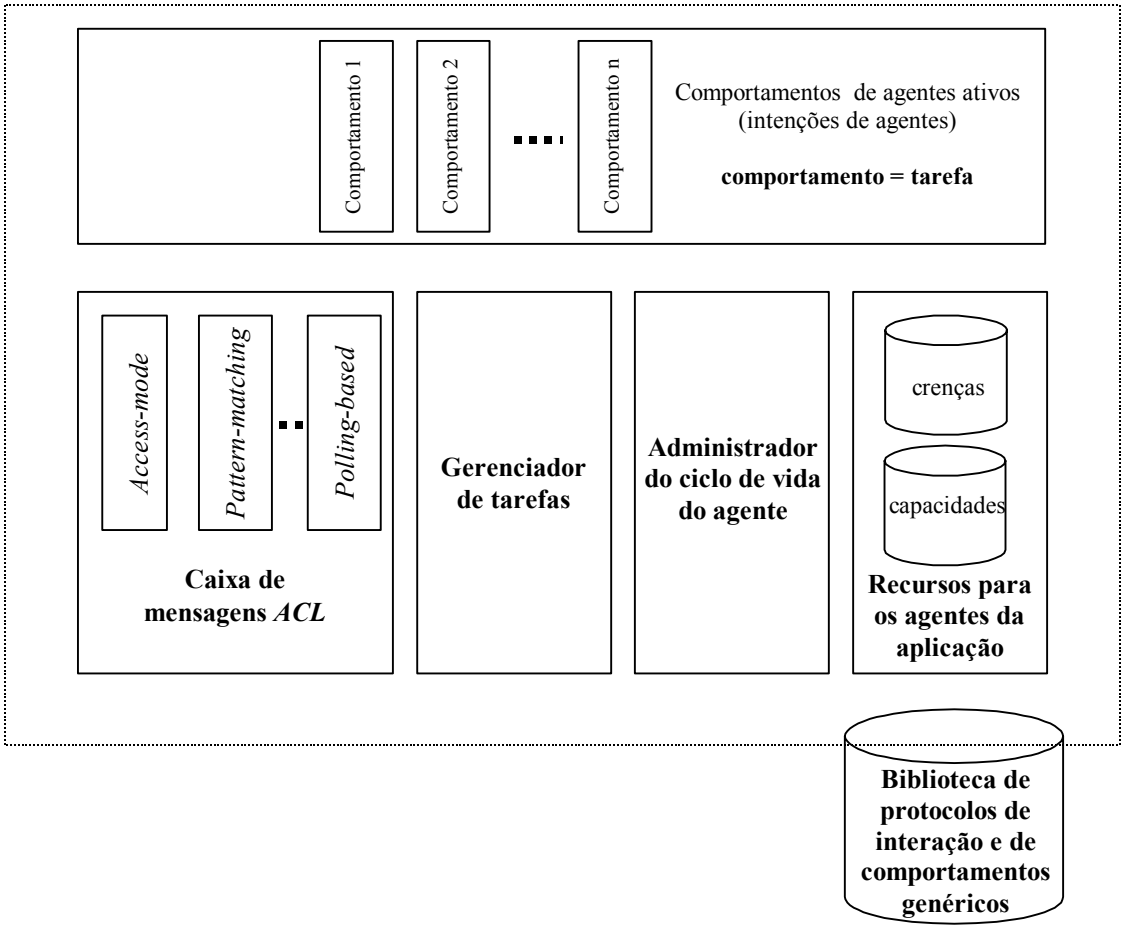

Figura 8 - Arquitetura do Agente Genérico da ferramenta *JADE* [33]

Um agente *JADE* pode apresentar vários estados, de acordo com o ciclo de vida do agente genérico especificado pela *FIPA*. Os estados são:

a) **Inicial**: o agente é construído, mas ainda não recebe registro no SGA; não possui nome nem endereço e não pode se comunicar com outros agentes;

b) **Ativo**: o agente é registrado no SGA, tem um nome e um endereço e pode acessar vários serviços *JADE*;

c) **Suspenso**: o agente é interrompido e nenhum comportamento é executado;

d) **Espera**: o agente é bloqueado, esperando por alguma mensagem;

e) **Destruído**: o agente morre e seu registro é destruído pelo SGA;

f) **Transitório**: o agente migra para um novo local; o sistema continua recebendo mensagens que serão depois envidas para o agente em seu novo endereço;

g) **Cópia**: estado interno usado por *JADE* durante a "clonagem" de um agente;

h) **Movido**: quando um agente móvel muda para um novo local.

O agente *JADE* provê métodos públicos para realizar transições entre os vários estados possíveis.

JADE não habilita agentes com capacidades específicas além daquelas necessárias para comunicação e interação. No entanto, o modelo de agentes permite a integração de programas externos com os agentes de tarefas. A biblioteca existente em *JADE* inclui uma classe, chamada *JessBehaviour*, que permite o uso da linguagem *JESS* (*Java Expert System Shell*) [40] como mecanismo de raciocínio do agente. *JESS* é bastante útil no controle da ativação e desativação dos comportamentos em *JADE*, o que permite a implementação de uma arquitetura formada por agentes reativos e deliberativos (onde *JESS* executa os papéis deliberativos e *JADE*, os reativos).

Em suma, *JADE* permite o desenvolvimento de sistemas de agentes aptos a:

a) Trabalhar de forma pró-ativa, de acordo com regras de uso;

b) Comunicar e negociar diretamente com outros agentes, indiferentemente do seu papel e posição;

c) Coordenar no sentido de resolver problemas complexos de forma distribuída.

No futuro, os desenvolvedores de *JADE* prometem o acréscimo de uma ferramenta visual para compor os comportamentos de agentes e oferecer uma arquitetura de mais alto nível.

3.4.2 *MAST*

A arquitetura da *MAST* [39] foi concebida como um *framework* distribuído para a cooperação de múltiplos agentes heterogêneos.

Esta arquitetura é constituída por duas entidades básicas: os agentes e a rede na qual eles interagem. As funcionalidades básicas e interfaces da rede formam o modelo de rede.

Um agente em *MAST* é estruturado como um conjunto composto pelos seguintes elementos:

a) **Serviços**. Funcionalidades oferecidas a outros agentes;
b) **Objetivos**. Tarefas de interesse próprio;
c) **Recursos**. Informações externas (bibliotecas, ontologias, etc);
d) **Objetos internos**. Estruturas de dados compartilhados por todos os processos que são disponibilizados pelo agente para responder a solicitações de serviços ou para alcançar seus objetivos;
e) **Controle**. Especificação de como os agentes lidam com as solicitações de serviços.

Várias primitivas de comunicação são implementadas, incluindo diferentes mecanismos de sincronização (síncrono, assíncrono e comunicação adiada) e protocolos de alto nível, como *Contract Net* [58], um padrão *FIPA*.

No nível de rede, um serviço de páginas amarelas é oferecido por um agente especializado, o agente *YP* (*Yellow Pages*). Ao serem criados, cada agente registra, no *YP*, seu endereço, os serviços que oferece e os serviços que deseja.

Os agentes podem ser divididos em grupos, que podem ser utilizados como *aliases* em petições de serviços. Desta forma, estes pedidos podem ser endereçados a um agente individual, a agentes de um grupo ou a todos os agentes que oferecem o serviço. O agente *YP* responde às solicitações fornecendo todas as informações que um agente precisa saber (p.ex., o endereço do agente que oferece um serviço). Tais informações são continuamente atualizadas pelo agente *YP*.

Um agente *YP* age como um tipo de repositório ativo, não como um roteador. Ele registra e difunde informação acerca do conjunto de agentes que cooperam em uma aplicação particular. Por esta razão, diferentes *YP*s podem estar ativos simultaneamente, inclusive na mesma máquina (contanto que eles usem portas distintas para comunicação).

A *MAST* utiliza uma linguagem especializada, a *MAST-ADL* (*MAST Agent Description Language*) [39], para simplificar a especificação dos agentes que cooperam na

resolução de um problema em um *framework* híbrido. Um arquivo *MAST-ADL* é formado por duas seções:

a) **Preâmbulo**. Cabeçalho contendo os seguintes itens: o endereço do *YP*, a linguagem-padrão que será usada para codificação de mensagens, as ontologias-padrão e bibliotecas usadas para implementação em baixo nível;

b) **Corpo**. Define a estrutura do conjunto de agentes; classes de agentes e instâncias podem ser declaradas numa hierarquia.

A *MAST* dispõe de ferramentas para transcrever os arquivos *ADL* em programas C++ [15] e, subseqüentemente, compilá-los.

Um dos destaques da arquitetura *MAST* é a possibilidade de se utilizar dois níveis de integração. Por padrão, os agentes da arquitetura são fracamente acoplados; isto significa que a comunicação entre agentes é feita via passagem de mensagem. No entanto, um mecanismo de acoplamento forte é necessário (principalmente para fins de eficiência) sempre que existe um fluxo contínuo de interação entre agentes.

3.4.3 *AgentTool*

AgentTool [17] é um ambiente gráfico de desenvolvimento escrito em Java para auxiliar usuários na análise, projeto e implementação de SMAs. Ele foi projetado para dar suporte à metodologia *MaSE*, descrita na Seção 2.1.4. A *MaSE* é usada pelo projetista para definir graficamente o comportamento do sistema em um alto nível (Figura 9).

Na fase de projeto do sistema são definidos os tipos de agentes, bem como as possíveis comunicações que podem ocorrer entre eles. Esta especificação em nível de sistema é então refinada para cada tipo de agente no sistema. Para refinar um agente, o desenvolvedor seleciona e cria uma arquitetura do agente para em seguida detalhar o seu comportamento.

Uma vez que o sistema tenha sido completamente especificado, o projetista gera o código para o SMA. Somente neste momento o projetista realmente define o *framework*, bem como qualquer implementação específica de comunicação e os protocolos de segurança. Estas definições são construídas automaticamente dentro do código durante a síntese do sistema.

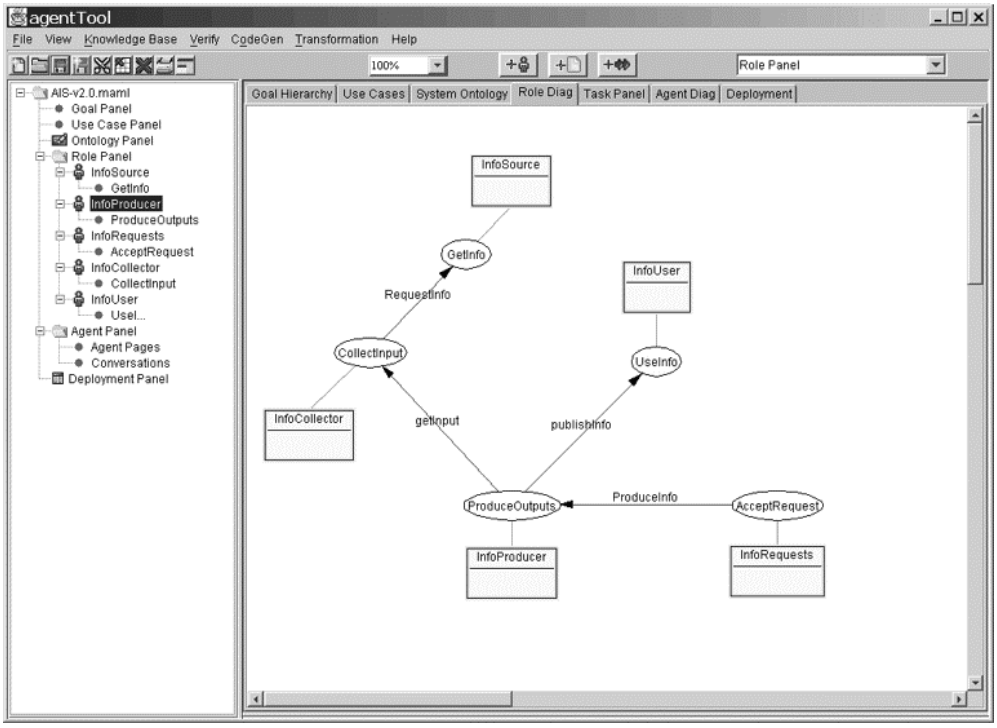

Figura 9 - Especificação de um diagrama de papéis em *AgentTool* [17]

AgentTool possui uma biblioteca de componentes de *software* pré-existentes. Cada componente tem uma definição formal que permite o usuário determinar se o componente satisfaz os requisitos para o agente especificado. Desta forma, o *AgentTool* precisa apenas sintetizar o código que não foi implementado via reuso.

O principal objetivo da *AgentTool* é aplicar técnicas na construção de agentes inteligentes para garantir a incorporação de protocolos de comunicação e segurança apropriados.

3.4.4 *JAFMAS*

JAFMAS [8] é um *framework* Java para representar e desenvolver protocolos e conhecimento de cooperação em um SMA. O *framework* permite que os agentes trabalhem juntos e alcancem coerentemente seus objetivos.

A maioria das ferramentas baseadas em agentes é essencialmente centrada na comunicação, sem definir o comportamento social dos agentes numa comunidade. Mesmo as que definem o modelo social de um agente, usam gerenciamento centralizado de agentes.

JAFMAS representa uma alternativa para o desenvolvimento de SMAs totalmente flexíveis, escaláveis, tolerantes a falhas e autoconfiguráveis.

JAFMAS provê um conjunto de serviços que auxiliam o desenvolvedor na implementação de :

a) Protocolos de comunicação;

b) Interação entre agentes;

c) Coerência e coordenação na agência.

A Figura 10 mostra a arquitetura completa da *JAFMAS*. As camadas mais baixas representam o *hardware* e o sistema operacional. O modelo local é específico para cada aplicação, cabendo ao usuário implementá-lo quando necessário. O uso de Java como infraestrutura no *framework* permite ao usuário integrá-lo ao legado existente de aplicações e bancos de dados, enquanto desenvolve seus modelos locais. Acima da infra-estrutura Java está a camada de comunicação formada, na verdade, por duas subcamadas: camada lingüística (linguagem comum para comunicação) e camada de protocolos (modos de comunicação empregados). Quando a comunicação entre os agentes é estabelecida, provisões devem ser feitas para que eles interajam um com os outros de forma coordenada e cooperativa. Acima da camada de comunicação está o modelo social do agente.

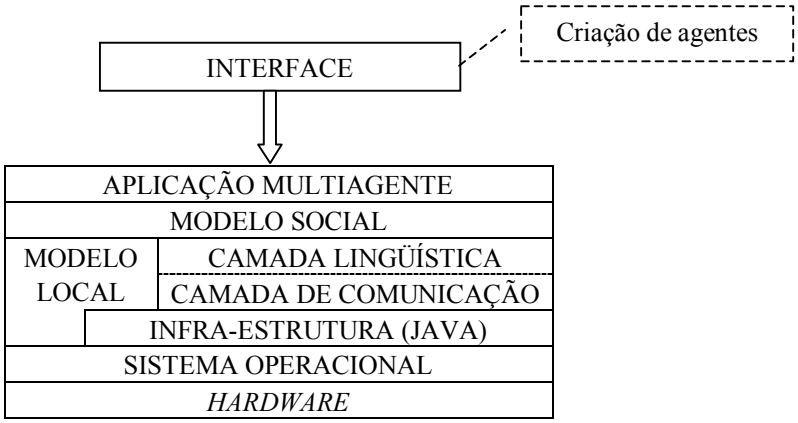

Figura 10 - Arquitetura da ferramenta *JAFMAS* [8]

Uma vez fornecidos os serviços de comunicação e coordenação, a ferramenta está pronta para construir agentes.

3.4.5 *PTK* e *AgentFactory*

A metodologia *PASSI*, descrita na Seção 2.1.4, é automatizada por duas ferramentas: *PTK* [13], que constitui-se de um *plugin* para a ferramenta *CASE* baseada em *UML Rational Rose* [3], e *AgentFactory* [10], uma aplicação de reuso de padrões que permite a rápida prototipação de grande parte de sistemas complexos em conformidade com *FIPA*.

O *PTK* permite ao usuário seguir o processo de análise e projeto da *PASSI* (Figura 11), provendo uma série de funcionalidades que são específicas para cada fase do processo, por meio de menus que aparecem sempre que um elemento (classe, caso de uso, etc) *UML* é selecionado. As principais funcionalidades do *PTK* estão descritas a seguir.

a) **Compilação automática de diagramas**. Permite ao projetista seguir ao longo dos passos da *PASSI* de forma rápida e fácil. Por exemplo, a identificação de agentes é totalmente feita pela ferramenta uma vez que o usuário tenha atribuído funcionalidades ao agente. A simples identificação de um agente dispara uma série de operações automáticas: construção do diagrama de especificação de tarefas, modelagem do esqueleto do agente, modelo descritivo do comportamento, etc;

b) **Suporte automático à execução de operações recorrentes**. É possível a modificação dos modelos a qualquer instante. Toda alteração feita dispara a atualização automática de todos os diagramas relacionados;

c) **Consistência de projeto**. A ferramenta realiza a checagem de modelos. Ao ser invocada, esta operação verifica a precisão e a consistência dos diagramas construídos até este ponto. Além disso, esta operação é executada sempre que o usuário completa uma fase. Qualquer erro ou inconsistência é assinalado;

d) **Compilação automática de relatórios e documentação**. O *PTK* pode produzir um relatório de todo modelo no formato *Microsoft Word®*;

e) **Geração de código e engenharia reversa**. Código pode ser gerado a partir dos diagramas do modelo de implementação. O código produzido é, na verdade, o esqueleto do agente escrito em Java, incluindo as subclasses de tarefas. Outrossim, este *plugin* permite a engenharia reversa pela criação de um diagrama de definição da estrutura do agente no modelo *Rose* a partir do código-fonte Java.

Durante esta operação, *PTK* atualiza todos os diagramas relacionados. Partes adicionais do código são automaticamente produzidos à medida que padrões são introduzidos.

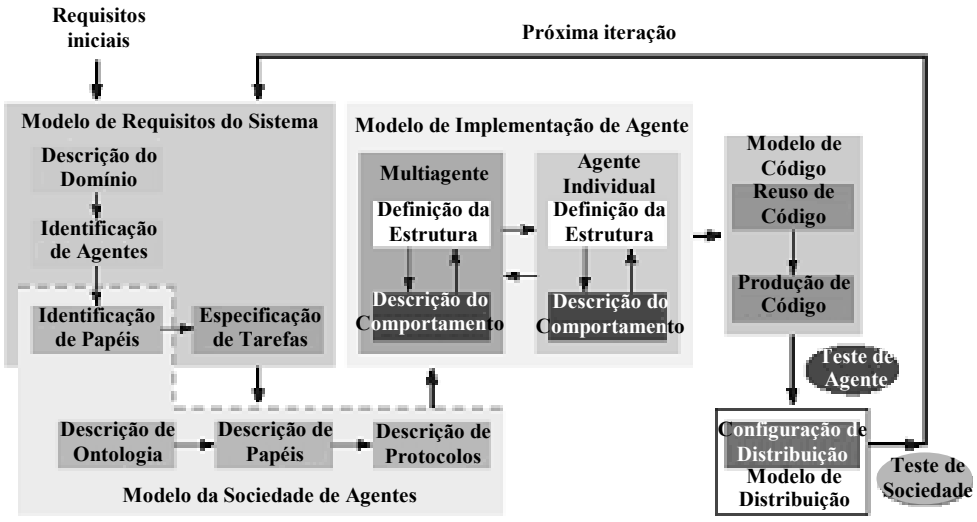

Figura 11 - Modelos da *PASSI* automatizados pelo *PTK* [13]

Os principais objetivos do PTK são: simplificar o trabalho dos projetistas de SMAs e permitir a geração automática de uma parte considerável do código.

A idéia por trás do PTK é que projetar um SMA corresponde a instanciar um metamodelo do sistema multiagente que preenche os requisitos do problema específico. Por esta razão, escolhas específicas (como designação de agentes e suas funcionalidades) têm conseqüência direta nos outros passos do processo. Esta abordagem faz com que os diferentes diagramas que constituem o resultado final do projeto realizado com o auxílio da *PASSI* sejam gradualmente compostos pelo projetista e pela sua interação com o *PTK*.

Alguns destes diagramas são totalmente dependentes do projetista, alguns são automaticamente feitos pela ferramenta e outros são construídos parcialmente pelo *PTK* e completados pelo desenvolvedor. O relatório gerado por este *plugin* contém, além dos diagramas, descrições textuais e algumas tabelas que resumem os agentes, seus comportamentos, papéis, ontologias, etc.

A outra ferramenta de suporte a *PASSI*, *AgentFactory*, pode, rapidamente, criar SMAs complexos usando um grande repositório de padrões e ainda prover a documentação do

projeto dos agentes. Esta ferramenta pode também trabalhar em linha como uma aplicação *Web*[10].

Os padrões usados resultam da composição de três diferentes aspectos de um SMA:

a) A estrutura estática de um ou mais agentes ou partes dele (i.e. comportamentos);

b) A descrição do comportamento dinâmico expresso pelos elementos previamente citados;

c) O código de programação que realiza tanto a estrutura estática (esqueletos) e o comportamento dinâmico (parte dos métodos internos) em um contexto específico da plataforma de agentes.

Como é sabido, a aplicação de um padrão a um sistema existente causa significativas mudanças em sua estrutura, e isso implica em outras modificações no sentido de harmonizar os novos elementos com os já existentes. O *AgentFactory* auxilia neste processo, completando os padrões por meio de uma coleção de restrições que descrevem como o sistema em questão deve se adaptar para aceitar corretamente as novas partes. O *AgentFactory* permite ainda a geração automática de códigos-padrão para a ferramenta *JADE*, descrita na Seção 3.4.1.

3.4.6 Tabela Comparativa

A Tabela 6 mostra as atividades que recebem o suporte das ferramentas discutidas nas seções anteriores. Ferramentas para geração de código, documentação, prototipação podem ser usadas ao longo do processo de desenvolvimento de *software*.

Algumas ferramentas, como *AgentTool*, *JAFMAS* e *PTK*, oferecem suporte da análise à implementação. Outras, apenas às fases iniciais do CVD*S* ou apenas à fase de implementação.

Em relação à funcionalidade das ferramentas, observa-se que a maioria enfatiza as atividades relacionadas à comunicação e cooperação, essenciais numa agência. Quanto ao suporte a métodos, a maioria possui métodos ou metodologias como guias de utilização.

[10] Disponível em: http://mozart.csai.unipa.it/af

Há casos em que uma ferramenta incrementa e/ou complementa a funcionalidade de outra, como ocorre com as ferramentas *PTK* e *AgentFactory*. A *PTK* é um *plugin* para a ferramenta *Rational Rose®* que visa aumentar a robustez e a coerência de projeto, além de diminuir o esforço do projetista e prover suporte para a fase de implementação com a geração automática de grande parte de código. A *AgentFactory*, por sua vez, é uma aplicação para o reuso de padrões que permite a rápida prototipação de partes de complexos sistemas compatíveis com as especificações *FIPA* e ainda provê a documentação dos agentes compostos.

FASES DO CVD*S*		**FERRAMENTAS**					
		JADE	*MAST*	*AgentTool*	*JAFMAS*	*PTK*	*AgentFactory*
Análise			●	●	●	●	
Projeto			●	●	●	●	
Implementação		●		●	●	●	●
ATIVIDADES / CARACTERÍSTICAS	Geração de código		●	●		●	●
	Modelagem e simulação	●			●	●	
	Comunicação e cooperação	●	●	●	●	●	
	Definição de comportamento			●	●	●	●
	Desenvolvimento para reuso		●	●		●	
	Desenvolvimento com reuso						●
	Processamento de linguagem	●	●			●	●
	Suporte a métodos			●	●	●	●
	Uso de ontologias		●			●	
	Diagramação					●	●
	Prototipação						●
	Documentação textual					●	●
	Engenharia Reversa					●	

Tabela 6 - Comparação entre algumas ferramentas para o desenvolvimento de SMAs

3.5 Considerações Finais

Neste capítulo foi apresentado um breve histórico das ferramentas *CASE*, bem como os benefícios de sua utilização e a principal classificação adotada na literatura especializada. Em seguida, foram mostrados os principais tipos de ambientes integrados, enfatizando a integração de ferramentas como fator fundamental para o sucesso da tecnologia *CASE*.

As ferramentas *CASE* sofreram inúmeras inovações ao longo dos tempos: da simples documentação e diagramação de sistemas, passando pela geração automática de código e pela prototipação, até a automatização de metodologias e suporte ao reuso e à reengenharia.

Ferramentas e ambientes estão tornando-se cada vez mais cruciais na Engenharia de *Software* à medida que a demanda por programas, bem como a diversidade de domínio e a complexidade aumentam. Os principais benefícios trazidos pelo uso das ferramentas assistentes são: provêem qualidade de *software*, facilitam o reuso e simplificam a manutenção de sistemas.

No entanto, a força da tecnologia *CASE* só é realmente alcançada quando as ferramentas encontram-se integradas em um ambiente. Com a integração, passa a existir melhorias no planejamento, na comunicação entre projetistas e no envio de informações de uma ferramenta para outra e de uma atividade a outra.

A integração pode ser parcial, reunindo ferramentas que dão suporte a parte do CVD*S*, ou total, que permite o auxílio a todo o processo de desenvolvimento. Existem três tipos de ambiente de desenvolvimento de *software*: ambiente de programação, *CASE workbench* e ambiente de Engenharia de *Software*.

Muitas propostas de métodos e representações tratam os sistemas baseados em agentes como sistemas orientados a objetos, porque muitos aspectos da OO servem para descrever agentes. Porém, é necessário que novas ferramentas e metodologias surjam para que as particularidades dos agentes de *software* sejam apropriadamente assistidas.

A Seção 3.4 discutiu algumas ferramentas para o desenvolvimento de sistemas multiagentes. A maioria oferece suporte às fases de análise, projeto e implementação e a atividades como geração de código, reuso, diagramação e documentação textual. Em comum, todas fornecem os serviços essenciais a uma sociedade de agentes, como comunicação, cooperação, coordenação e definição de comportamento.

Algumas das ferramentas discutidas são apropriadas ao desenvolvimento *com* reuso (Engenharia de Aplicações Multiagentes), porém nenhuma das analisadas fornece suporte para o desenvolvimento *para* reuso (Engenharia de Domínio Multiagente). Isso motivou a realização deste trabalho, com a concepção do *ONTOCADE*, um ambiente para análise e projeto de domínio.

No próximo capítulo será abordada a *MADEM*, uma metodologia para análise e projeto na Engenharia de Domínio Multiagente. Esta metodologia será automatizada pela ferramenta *ONTOCADE* para guiar a execução das atividades de análise e projeto da EDMA.

4 *MADEM*: UMA METODOLOGIA PARA ENGENHARIA DE DOMÍNIO MULTIAGENTE

Este capítulo apresenta a *MADEM* [20], uma metodologia que visa facilitar a execução das fases de análise e projeto na Engenharia de Domínio Multiagente. *MADEM* (Figura 12) integra e refina as técnicas *GRAMO* (*Generic Requirement Analysis Method based on Ontologies*) [21] e *DDEMAS (Domain Design for Multi-Agent Systems)* [23].

A *GRAMO* guia a captura e especificação de requisitos de uma família de sistemas, em um domínio de aplicação, para a geração do modelo de domínio. A *DDEMAS* guia a captura e especificação do projeto reutilizável para a geração de um *framework* multiagente.

A Tabela 7 apresenta as fases, tarefas e produtos da metodologia *MADEM* que são descritos nas próximas seções deste capítulo.

FASES	TAREFAS	SUBTAREFAS	PRODUTOS
Análise de Domínio	Modelagem de Domínio	Modelagem de Objetivos	*Modelo de Domínio* (Modelo de Objetivos, Modelo de Papéis, Modelo de Pacotes, Modelo de Interações entre Papéis e Modelo de Conceitos)
		Modelagem de Papéis	
		Modelagem de Variabilidades	
		Modelagem de Pacotes	
		Modelagem de Interações entre Papéis	
		Modelagem de Conceitos	
Projeto de Domínio	Projeto Arquitetural	Modelagem de Agentes	*Modelo de Projeto Arquitetural* (Modelo de Agentes, Modelo de projeto do *framework* e Modelo de atividades)
		Modelagem do *framework*	
		Modelagem de atividades	
	Projeto Detalhado	Refinamento dos agentes	*Modelo de Projeto de Domínio Detalhado* (Modelo de Comportamento dos Agentes e Modelo de interações entre agentes)
		Modelagem do Comportamento	
		Modelagem de interações entre agentes	

Tabela 7 - Fases, tarefas e produtos da *MADEM*

Conforme a Figura 12, a fase de análise da *MADEM* utiliza um sistema de padrões como insumo e produz um modelo de domínio, de acordo com o conhecimento do domínio ou problema. Já a fase de projeto utiliza o modelo de domínio gerado para produzir os modelos de projeto *arquitetural* e *detalhado*, baseados no sistema de padrões já existentes.

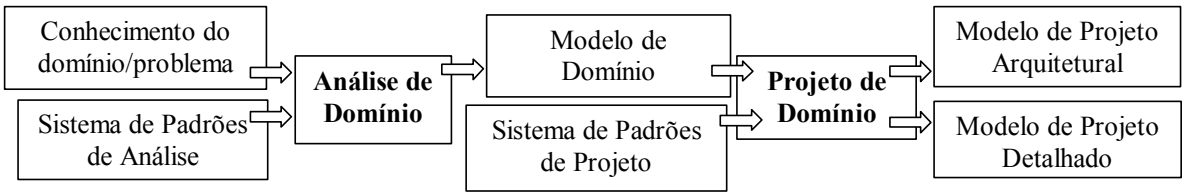

Figura 12 - Os insumos e os produtos da metodologia *MADEM*

4.1 Análise de Domínio

A fase de *Análise de Domínio* deve começar com a identificação do que será modelado: se a formulação de um problema ou de uma área de conhecimento. Esta fase consiste de uma única tarefa, a *Modelagem de Domínio*, que por sua vez, é composta por sete subtarefas: Modelagens de Objetivos, de Papéis, de Variabilidades, de Pacotes, de Interações entre Papéis, de Conceitos e de Axiomas.

Quando a formulação de um problema é modelada, são executadas apenas as tarefas de modelagem de objetivos, de papéis, de interações entre papéis e de variabilidades. Neste caso, o produto da modelagem é um modelo de domínio composto por modelos de objetivos, de papéis e de interações entre papéis.

Quando uma área de conhecimento é modelada, somente as tarefas de modelagem de conceitos e de variabilidades são executadas. Neste caso, o produto da modelagem é o modelo de domínio representado unicamente pelo modelo de conceitos.

Na *Modelagem de Objetivos*, inicialmente identifica-se o objetivo geral, tendo em vista o problema que os sistemas de uma família se propõem a resolver. Em seguida, são identificados os objetivos específicos, especializando-se ou refinando-se o objetivo geral. Por último, são identificadas as responsabilidades associadas a cada objetivo específico. O produto desta subtarefa é um *modelo de objetivos*, formado pelo objetivo geral, objetivos específicos e pelas responsabilidades. O modelo de objetivos é um diagrama de três níveis: o primeiro nível representa o objetivo geral, o segundo nível dispõe os objetivos específicos e o terceiro, é composto pelas responsabilidades (Figura 13).

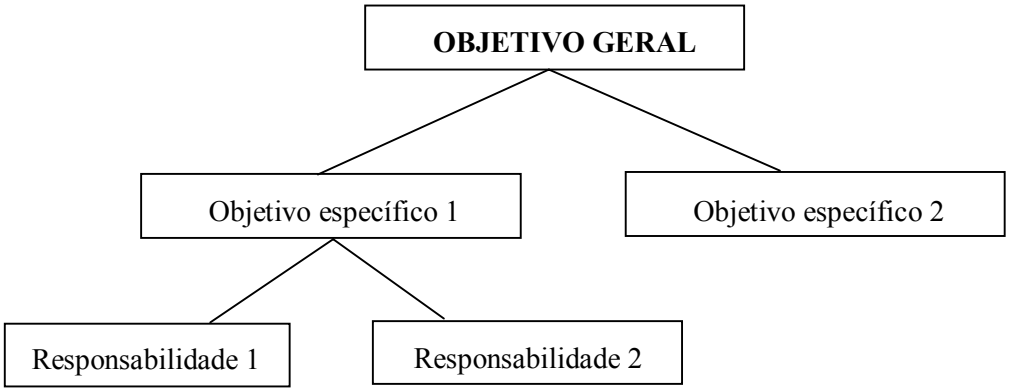

Figura 13 - Modelo de Objetivos definido pela *MADEM*

Na *Modelagem de Papéis*, cada responsabilidade identificada no modelo de objetivos é associada a um papel interno. Definem-se ainda as atividades que permitem o exercício de cada responsabilidade. Há situações em que uma atividade ou uma série de atividades relacionadas é executada por vários papéis. Neste caso, um papel independente deve ser criado. Em seguida, identificam-se as entradas, as saídas, os estados e os recursos que cada papel usa para a realização de suas atividades. O produto desta subtarefa é um *modelo de papéis* composto pelos papéis do sistema com suas respectivas responsabilidades, atividades, entradas, saídas, estados e recursos. O modelo de papéis é um diagrama de quatro níveis: o primeiro nível representa o papel; o segundo nível representa a responsabilidade do papel; o terceiro, agrupa as atividades do papel e o quarto nível é composto pelos recursos do papel (Figura 14).

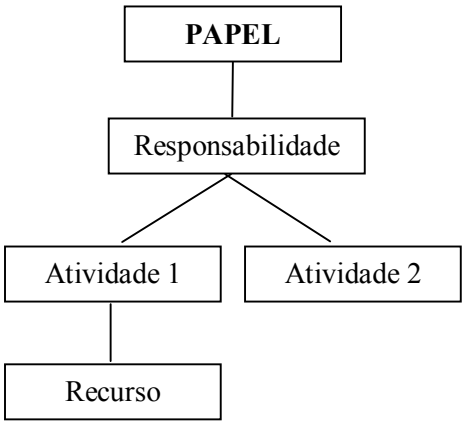

Figura 14 - Modelo de Papéis definido pela *MADEM*

Na *Modelagem de Variabilidades*, as instâncias dos conceitos (objetivos geral e específicos, responsabilidades, papéis, atividades e recursos) são classificadas em fixas e variáveis. Conceitos fixos representam as características comuns a todos os integrantes de uma família de sistemas, enquanto que, conceitos variáveis são aqueles que podem estar presentes ou não em sistemas de uma família e representam suas características particulares. Os conceitos são classificados de acordo com as regras apresentadas a seguir:

a) **O objetivo geral é fixo**. De acordo com o modelo de objetivos, um objetivo geral deve ser definido e alcançado por todos os sistemas de uma família de sistemas. Sendo assim, o objetivo geral é um conceito fixo.

b) **Os objetivos específicos são variáveis**. De acordo com o que é definido no modelo de objetivos, para que o objetivo geral seja alcançado, pelo menos um dos objetivos específicos tem que ser alcançado. Sendo assim, um objetivo específico é um conceito que pode estar presente ou não no desenvolvimento de cada sistema e, portanto, é variável.

c) **As atividades podem ser fixas ou variáveis**. Os papéis exercem suas responsabilidades por meio da execução de atividades. Atividades são fixas se elas devem fazer parte de todos os sistemas da família. Do contrário, as atividades são variáveis.

d) **As responsabilidades podem ser fixas ou variáveis**. Os objetivos específicos são alcançados com o exercício de suas responsabilidades. Se a responsabilidade contribui para o alcance de todos os objetivos específicos e é exercida pelas atividades fixas, então a responsabilidade é classificada como fixa. Do contrário, as responsabilidades são variáveis.

e) **Os papéis podem ser fixos ou variáveis**. As responsabilidades são exercidas pelos papéis. Então os papéis são classificados de acordo com suas responsabilidades: papéis são variáveis se exercem responsabilidades variáveis. Caso contrário, são fixos.

f) **Os recursos podem ser fixos ou variáveis**. Para a execução de atividades, os papéis dispõem de recursos. Um recurso é classificado como fixo se for requerido por pelo menos uma atividade fixa. Caso contrário, o recurso é classificado como variável.

g) **Os conceitos do domínio podem ser fixos ou variáveis**. Os conceitos do domínio que estão presentes em todos os sistemas da família são classificados como fixos. Do contrário, os conceitos do domínio são variáveis.

Na *Modelagem de Pacotes*, é feito um agrupamento dos papéis variáveis que cumprem com determinado objetivo específico. Para cada objetivo específico existirá um pacote contendo os papéis variáveis que auxiliam no cumprimento do objetivo específico. O produto desta subtarefa é o *Modelo de Pacote* que representa um pacote que contém papéis variáveis (Figura 15).

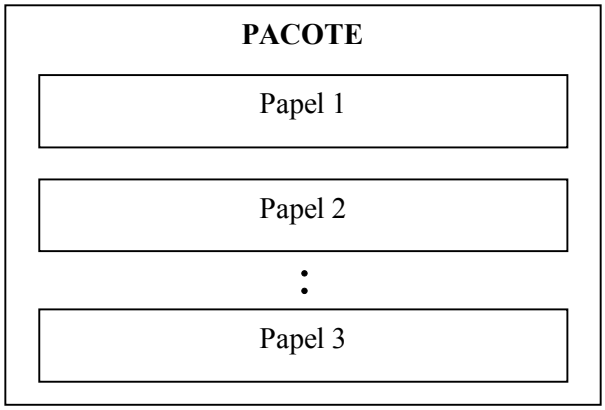

Figura 15 - Modelo de Pacote definido pela *MADEM*

Na *Modelagem de Interações entre Papéis*, são identificadas as interações que ocorrem entre os papéis e as entidades externas ao sistema. Para cada objetivo específico identificado no modelo de objetivos, é construído um modelo de interações entre papéis. Este modelo é o produto desta subtarefa, onde as interações são especificadas de forma seqüencial.

Na *Modelagem de Conceitos*, fontes de informação específicas (livros, revistas, artigos e relatórios) são analisadas e especialistas são consultados para permitir a identificação dos conceitos relevantes do domínio. Em seguida, é feita uma análise de aplicações existentes no referido domínio para a coleta de enfoques comuns a todas elas. O *modelo de conceitos*, produto gerado nesta subtarefa, captura a estrutura conceitual do domínio, permite um entendimento completo do domínio relacionado, oferece uma terminologia comum ao domínio sem ambigüidades, facilitando a comunicação. Esta modelagem pode valer-se dos padrões de análise que representam modelos definidos para determinados contextos, já testados e adotados em outros projetos. Desta forma, a modelagem de conceitos só precisa

detectar o que é necessário acrescentar a um padrão de análise ou como combinar esse padrão com outros para obter um modelo de conceitos completo. O modelo de conceitos é representado graficamente por uma rede semântica onde os nós representam os conceitos e as ligações representam os relacionamentos entre estes conceitos.

4.2 Projeto de Domínio

A fase de *Projeto de Domínio* tem, como entradas, modelos de domínio e sistemas de padrões. Esta fase consiste de duas tarefas: *Projeto Arquitetural* e *Projeto Detalhado*. O produto desta fase é um *framework* composto pelos modelos de *projeto arquitetural* e de *projeto detalhado*.

A tarefa de *Projeto Arquitetural* consiste das seguintes subtarefas: modelagem de agentes, modelagem do *framework* e modelagem de atividades. Os produtos desta tarefa são o modelo de agentes, modelo de projeto do *framework* e modelo de atividades.

Na *Modelagem de Agentes* são identificados os agentes que irão compor o modelo de projeto do *framework*. Os agentes são identificados a partir dos papéis especificados no modelo de papéis. Tem-se como proposta inicial o mapeamento de um "papel" para um "agente", no entanto, os agentes e papéis podem possuir um relacionamento do tipo "*1..n* para *1..n*". Cada agente irá realizar um conjunto de atividades necessárias para o cumprimento de suas responsabilidades. O produto desta subtarefa é um modelo de agente composto pelos agentes e pelos seus papéis desempenhados. O modelo de agentes é um diagrama de dois níveis: o primeiro nível é composto pelo agente e o segundo nível é composto pelo(s) papel(éis) correspondente(s) (Figura 16).

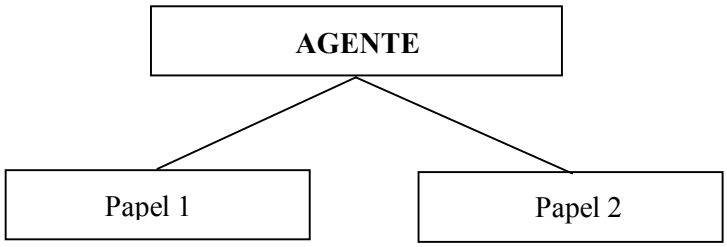

Figura 16 - Modelo de Agentes definido pela *MADEM*

A *Modelagem do framework* tem o propósito de integrar os agentes modelados para formar uma solução computacional ao problema tratado. Estes agentes devem ser organizados de forma adequada, a fim de que o sistema resolva eficientemente o problema. O produto desta subtarefa é um modelo de projeto do *framework*. Para construí-lo, é necessário realizar os seguintes passos: construção do esboço do projeto do *framework*, seleção de padrão arquitetural e construção do modelo de projeto de *framework*.

No esboço do projeto do *framework* organizam-se os agentes identificados na modelagem de agentes de forma a colaborarem entre si para solucionar o problema.

A seleção do padrão arquitetural é feita baseada no esboço do projeto do *framework*. Esta seleção fornece informações necessárias para a estruturação do SMA, enfatizando os aspectos de coordenação e cooperação entre os agentes da sociedade. A seleção do padrão arquitetural é feita da seguinte forma:

a) Analisa-se o problema que se pretende resolver identificando-se as características ou pontos determinantes e em seguida, compara-se com os padrões disponíveis;
b) Seleciona-se um ou mais padrões adequados ao problema e contexto abordados.

A construção do modelo de projeto do *framework* inicia-se após a seleção do padrão arquitetural com a organização da sociedade multiagente de acordo com a solução proposta pelo(s) padrão(ões). É estabelecida, então, uma estrutura organizacional para a sociedade multiagente, aplicando-se a solução descrita no padrão. São estabelecidos os mecanismos de coordenação e cooperação a serem utilizados na agência, que também são encontrados no padrão selecionado. O modelo de projeto do *framework* é inspirado no diagrama de colaboração da *AUML* [44].

A Modelagem de Atividades representa as atividades dos agentes e suas interações, obtidas a partir do modelo de agentes. Constrói-se um modelo de atividades que é inspirado no diagrama de atividades da *AUML* [44].

A tarefa de *Projeto Detalhado* é formada pelas seguintes subtarefas: refinamento dos agentes, modelagem de comportamento e modelagem de interações entre agentes. Os produtos desta tarefa são os modelos de comportamento e de interações entre agentes.

No Refinamento dos Agentes, o objetivo é o detalhamento de cada um dos agentes que compõem o modelo de projeto do *framework*. Para o refinamento dos agentes é

necessário realizar os seguintes passos: seleção do padrão de *projeto detalhado* e classificação dos agentes.

Na *Seleção do Padrão de Projeto Detalhado*, escolhe-se o padrão de *projeto detalhado* que fornece informações necessárias para a especificação do agente. A seleção do padrão de *projeto detalhado* é feita da seguinte forma:

a) Analisa-se o problema que o agente pretende resolver;

b) Seleciona-se um ou mais padrões disponíveis que combinem com o problema e o contexto tratados, além de analisar as forças[11] destes padrões.

A Classificação dos Agentes identifica os tipos de agentes segundo os dois tipos básicos: reativo e deliberativo. Um agente reativo é uma entidade que age usando um tipo de comportamento baseado em estímulo/resposta, seu comportamento é reduzido à execução de um conjunto de regras de condição-ação (do tipo se...então). Um agente deliberativo é uma entidade que possui um modelo simbólico interno. Ele elabora e seleciona planos ou ações, por meio de um processo baseado em raciocínio lógico, para que possa atingir sua meta no contexto da situação em questão.

Na Modelagem de Comportamentos, o comportamento do agente é identificado por meio do modelo de atividades, onde as atividades executadas pelo agente definem o seu comportamento. Tem-se como proposta inicial o mapeamento de uma "atividade" para um "comportamento", no entanto, atividades e comportamentos podem apresentar relações to tipo "*1..n para 1..n*". O modelo de agentes refinado é inspirado no diagrama de estado da *AUML* [44], que é composto pelos estados e pelos comportamentos que disparam as transições de estado (Figura 17).

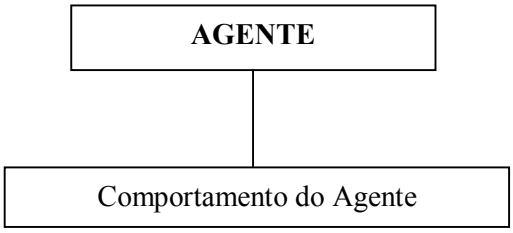

Figura 17 - Modelo de Comportamento do Agente definido pela *MADEM*

[11] Atributos que mostram os fatores que influenciam a solução proposta para um padrão e como ela é implementada. Também podem apresentar as restrições da solução, as desvantagens de sua utilização e as razões de sua escolha em detrimento a outras estratégias.

Na Modelagem de Interações entre Agentes, as interações que ocorrem entre agentes internos e externos são ilustradas na ordem em que ocorrem. O modelo de interações entre agentes é inspirado no diagrama de seqüência da *UML*.

4.3 *ONTOMADEM*: Conhecimento da metodologia *MADEM*

Esta seção apresenta a *ONTOMADEM* (*MADEM Generic Ontology*) [20], uma ontologia genérica que captura o conhecimento da metodologia *MADEM*. O *ONTOCADE* utiliza esta ontologia para integrar as ferramentas que o compõem em torno das atividades de análise e projeto de domínio.

Como visto na Seção 2.3, a captura de conhecimento é feita apropriadamente pelos editores de ontologias. A construção da *ONTOMADEM* foi feita com o auxílio do *Protégé* [55].

O processo de construção da *ONTOMADEM* (Figura 18) consiste em duas fases: a definição e o projeto da ontologia. Na fase de definição é utilizado o conhecimento da *MADEM* para gerar uma rede semântica com a representação dos conceitos da metodologia. Na fase de projeto, a *ONTOMADEM* é criada mapeando-se a rede semântica a uma ontologia representada por uma hierarquia de classes.

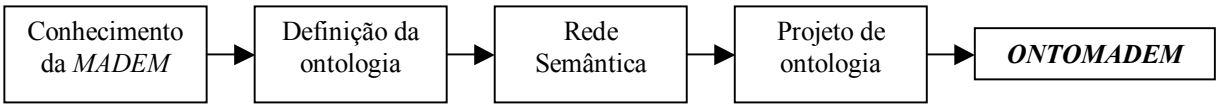

Figura 18 - Processo de construção da *ONTOMADEM* [20]

4.3.1 Definição da ontologia

Na fase de definição da ontologia, o conhecimento da *MADEM* é representado em redes semânticas (Figuras 21 a 24).

A Figura 19 mostra parte da rede semântica que representa os conceitos de modelagem, seus relacionamentos e atributos: objetivo, papel, responsabilidade, atividade, recurso, interação, entidade externa, agente, comportamento, padrão, demais conceitos do domínio. As setas rotuladas mostram como os conceitos relacionam-se entre si, de acordo

com as regras comumente existentes nas técnicas de análise de requisitos e de projeto. Por exemplo, um objetivo geral é um "tipo de" objetivo; um papel interno "compõe" um pacote; um agente "desempenha" um papel; etc.

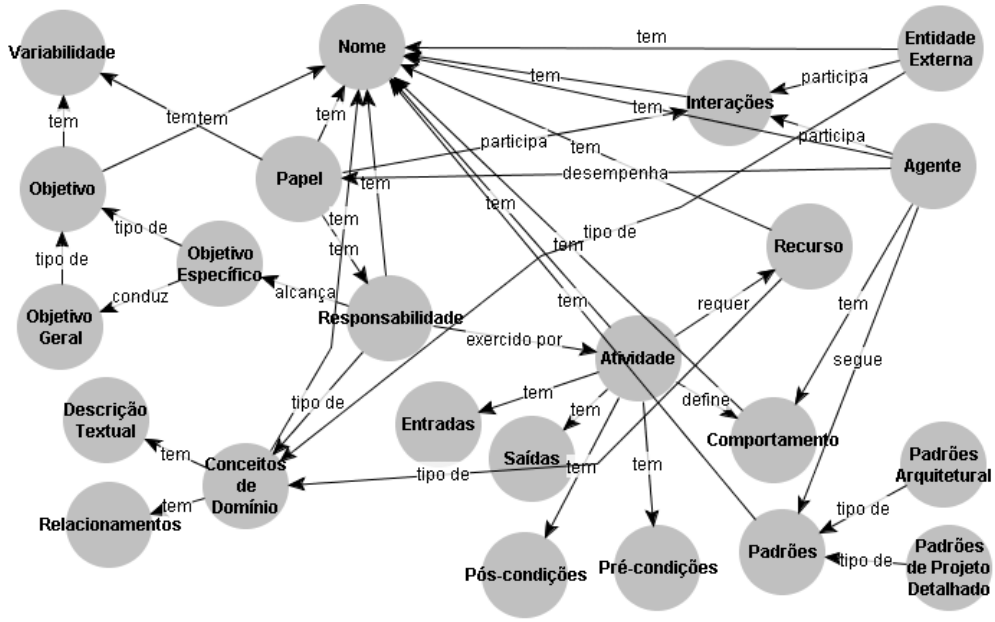

Figura 19 - Rede Semântica dos Conceitos de Modelagem

A Figura 20 mostra parte da rede semântica que representa a tarefa de modelagem de domínio.

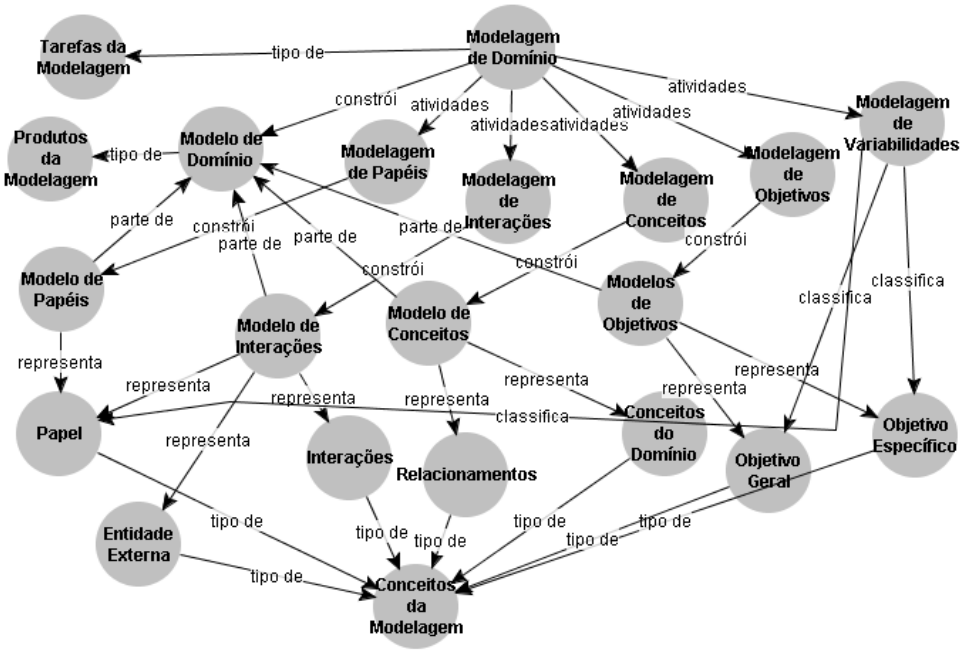

Figura 20 - Rede Semântica da Tarefa de Modelagem de Domínio

A Figura 21 mostra parte da rede semântica que representa a tarefa de *projeto arquitetural*.

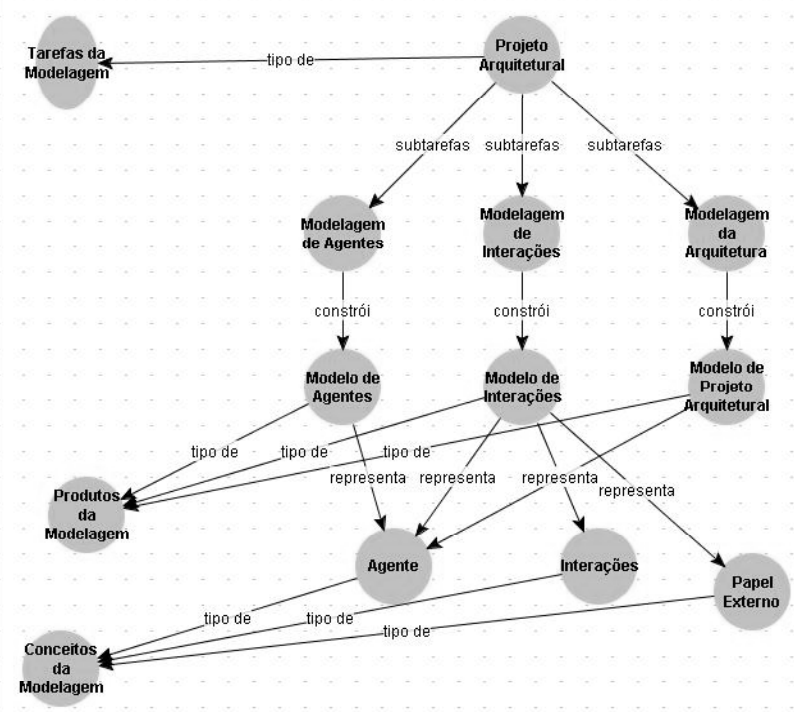

Figura 21 - Rede Semântica da Tarefa de Projeto Arquitetural

A Figura 22 mostra parte da rede semântica representando a tarefa de *projeto detalhado*.

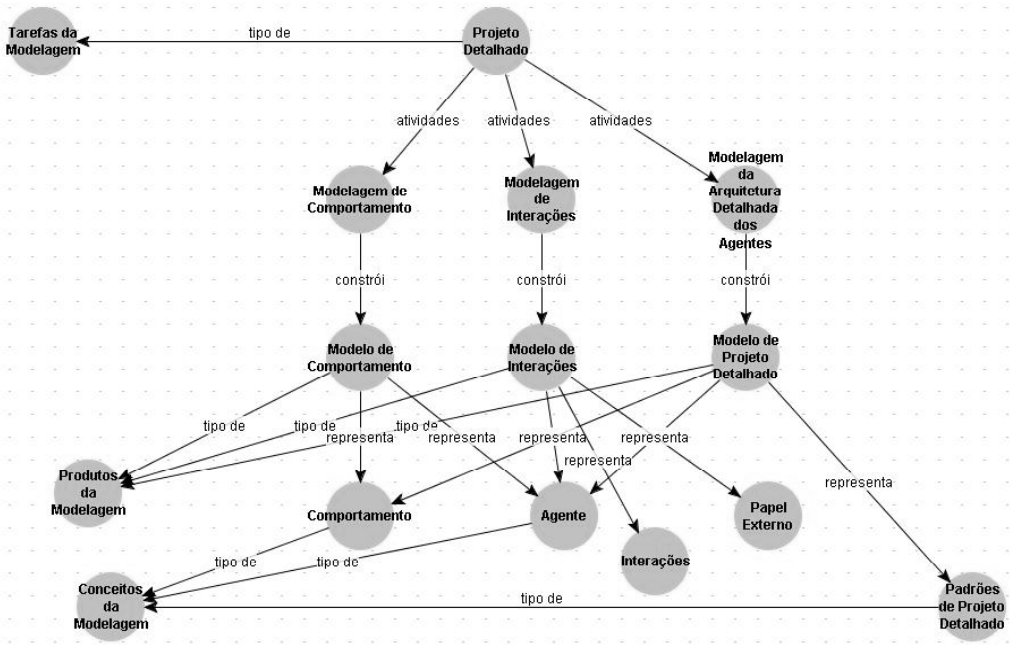

Figura 22 - Rede Semântica da Tarefa de Projeto Detalhado

4.3.2 Projeto da ontologia

Na fase de projeto da ontologia, os conceitos e relacionamentos da rede semântica são mapeados para compor a *ONTOMADEM*. Os nós são mapeados para metaclasses e aqueles que possuem um relacionamento rotulado "tipo de" são mapeados em uma hierarquia de classes. Outros relacionamentos são mapeados para *slots* das correspondentes classes. Cada *slot* é associado com facetas apropriadas como tipo e cardinalidade.

A Figura 23 mostra a hierarquia de metaclasses da *ONTOMADEM* e, em destaque, a metaclasse modelagem de domínio.

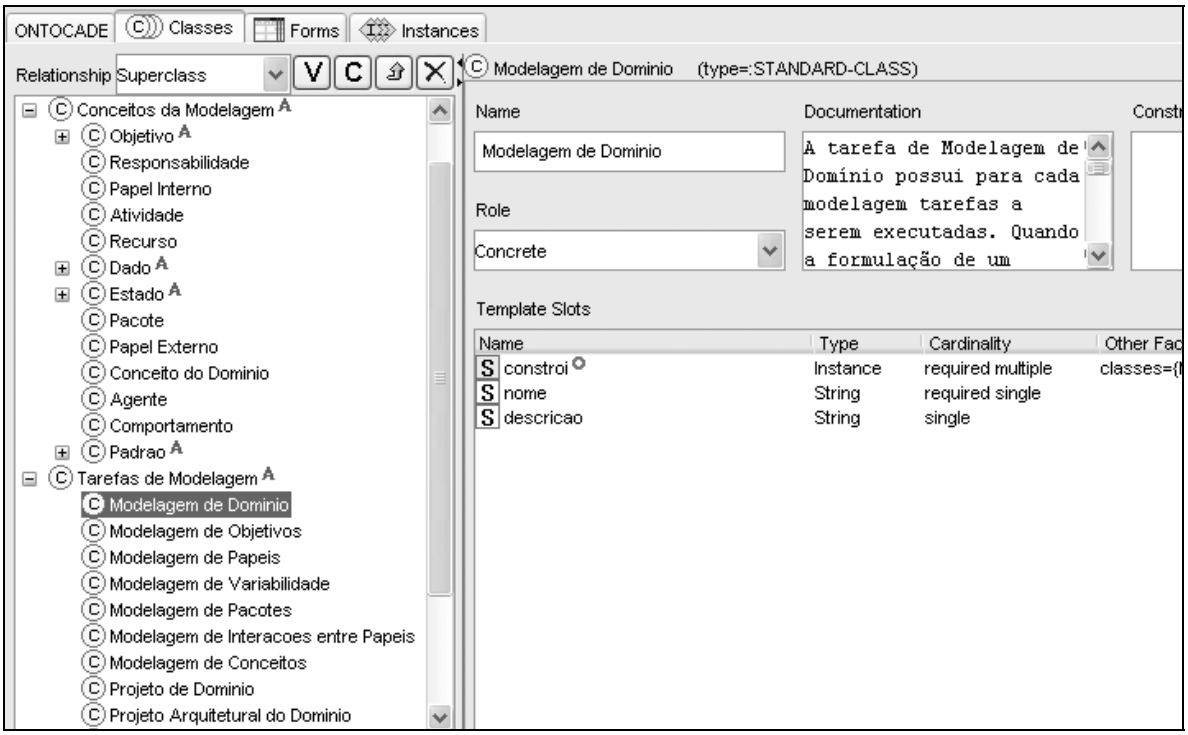

Figura 23 - Hierarquia de meta-classes e a meta-classe "modelagem de domínio"

Metaclasses são parte do modo como *Protégé* manipula e cria classes internamente. *Protégé* usa as metaclasses para construir as classes específicas e determinar suas propriedades. Todos os *slots* criados para uma metaclasse aparecem como *widget* para qualquer classe que use a metaclasse.

4.4 Considerações Finais

Este capítulo apresentou a *MADEM* [20], uma metodologia que visa facilitar a execução das fases de análise e projeto na Engenharia de Domínio Multiagente. A *MADEM* surgiu da necessidade de se integrar e refinar as técnicas *GRAMO* [21] e *DDEMAS* [23].

As principais alterações feitas na *GRAMO* durante o processo de concepção da *MADEM* foram:

a) Tornou-se mais clara a distinção entre especificação de problemas e área de conhecimento;

b) Alterou-se a ordem de execução das tarefas de modelagem, que passou a ser em função do que será modelado;

c) Novos conceitos foram introduzidos;

d) Introduziram-se novas tarefas de modelagem: a de variabilidades, de axiomas e de pacotes.

As maiores alterações foram feitas à técnica *DDEMAS*, que foi praticamente reescrita.

O conhecimento de uma metodologia é essencial para que a ferramenta auxilie eficientemente as fases do CVD*S*, seja parcial ou totalmente. Com a incorporação do conhecimento da metodologia, a ferramenta garante a execução correta das atividades do processo de desenvolvimento. A captura do conhecimento foi feito com o *Protégé*, que permitiu a geração da ontologia genérica *ONTOMADEM* [20].

Na *ONTOMADEM*, todos os conceitos da *MADEM* foram dispostos em uma hierarquia de metaclasses, a partir das quais novas classes podem ser criadas para representam o domínio de uma aplicação específica.

O próximo capítulo apresentará o *ONTOCADE*, uma proposta de ambiente *CASE* para análise e projeto de domínio multiagente.

5 *ONTOCADE*: UM AMBIENTE *CASE* BASEADO EM ONTOLOGIAS PARA ANÁLISE E PROJETO NA ENGENHARIA DE DOMÍNIO MULTIAGENTE

Este capítulo introduz o *ONTOCADE*, um ambiente *CASE* baseado em ontologias para análise e projeto na EDMA. *ONTOCADE* fornece suporte à aplicação da *MADEM* [20], metodologia discutida no capítulo 4.

ONTOCADE é uma experiência pioneira, uma vez que não existem ferramentas destinadas à análise e projeto na EDMA, conforme foi possível constatar nas pesquisas realizadas em fontes especializadas no paradigma computacional baseado em agentes e relatadas no capítulo 3.

A Seção 5.1 enumera os requisitos do *ONTOCADE*. Na Seção 5.2 são mostrados os detalhes de concepção do ambiente. A Seção 5.3 faz o esboço do projeto, mostrando como as atividades da *MADEM* são automatizadas pelo *ONTOCADE*. A Seção 5.4 mostra os detalhes da implementação do *plugin* que será responsável pela ligação do *Protégé* com as ferramentas (interface e modeladores) do *ONTOCADE*. Na Seção 5.5 são mostradas as diretrizes de utilização do *plugin*.

5.1 Requisitos

Em suma, o *ONTOCADE* deve auxiliar o usuário (projetista) desde a captura de conhecimento do domínio até a construção do modelo do *framework* do sistema. Todo este processo é simplificado pela *MADEM*, que define as regras de análise (construção do modelo de domínio a partir do conhecimento de domínio) e projeto (construção dos modelos de *projeto arquitetural* e *detalhado* a partir do modelo de domínio).

A fim de dar suporte, de forma eficiente, às atividades da *MADEM*, o *ONTOCADE* precisa atender aos seguintes requisitos:

a) Facilitar a comunicação entre os projetistas por meio do compartilhamento de ontologias;

b) Fornecer meios para captura de requisitos na fase de análise de domínio;

c) Garantir controle rigoroso da aplicação das regras da *MADEM*;

d) Facilitar o reuso de padrões existentes;

e) Guiar a criação e atualização dos modelos do sistema: *Modelo de Domínio* (Modelo de Objetivos, Modelo de Papéis, Modelo de Pacotes, Modelo de Interações entre Papéis, Modelo de Conceitos), *Modelo de Projeto Arquitetural* (Modelo de Agente, Modelo do *framework* e Modelo de Atividades) e *Modelo de Projeto Detalhado* (Modelo de Comportamento dos Agentes e Modelo de Interações entre Agentes);

f) Disponibilizar um mecanismo de compartilhamento e gerenciamento de informações;

g) Simplificar as atividades de documentação e geração de relatórios;

h) Facilitar o acesso e utilização das ferramentas do ambiente;

i) Permitir extensões para a introdução de novas ferramentas.

As seções seguintes mostram como cada um destes requisitos são atendidos pelo *ONTOCADE*.

5.2 Concepção do ambiente

O ambiente *ONTOCADE* é orientado tanto a *função* (facilita a modelagem, oferece suporte a métodos e diagramação, por exemplo), como a *atividade* (fornece suporte às fases de análise e de projeto). Por esta razão, o ambiente possui a configuração de um *CASE workbench*, onde as ferramentas que o compõem estão integradas em torno de um RCI para fornecer suporte às fases iniciais da EDMA (Figura 24). Além disso, como será mostrado ao longo deste capítulo, o *ONTOCADE* apresenta os três tipos principais de integração: *por dados* (facilita o compartilhamento de informação), *por interface* (padronização da interação usuário-ambiente) e *por atividade* (incorporação do modelo de conhecimento da *MADEM*).

O ambiente possui oito tipos de ferramentas: Editor de texto, Gerador de relatórios, Dicionário de Dados, Mecanismo de Consultas, RCI, Modeladores, Ferramenta

para criação de formulários e Facilidades de importação / exportação. A seguir são descritas as peculiaridades destas ferramentas.

a) **Editor de Texto**. Permite a especificação de comentários a serem inseridos nos modelos;

b) **Gerador de Relatórios**. Gera sumário de descrições de toda ontologia criada. Cada relatório exibe a hierarquia dos *frames* que compõem a ontologia;

c) **RCI e DD**. Responsáveis pelo armazenamento e pela organização das informações relacionadas ao desenvolvimento de uma aplicação;

d) **Mecanismo de consultas**. Permite a seleção de elementos do projeto;

e) **Editor e visualizador de gráficos**. Permite a edição e a visualização dos diagramas que representam os modelos de domínio e de projeto;

f) **Facilidades de importação e exportação**. Permitem o compartilhamento de ontologias ou conversões entre os formatos possíveis de representação das mesmas;

g) **Modeladores**. Conjunto de ferramentas responsáveis pela modelagem de domínio e de projeto.

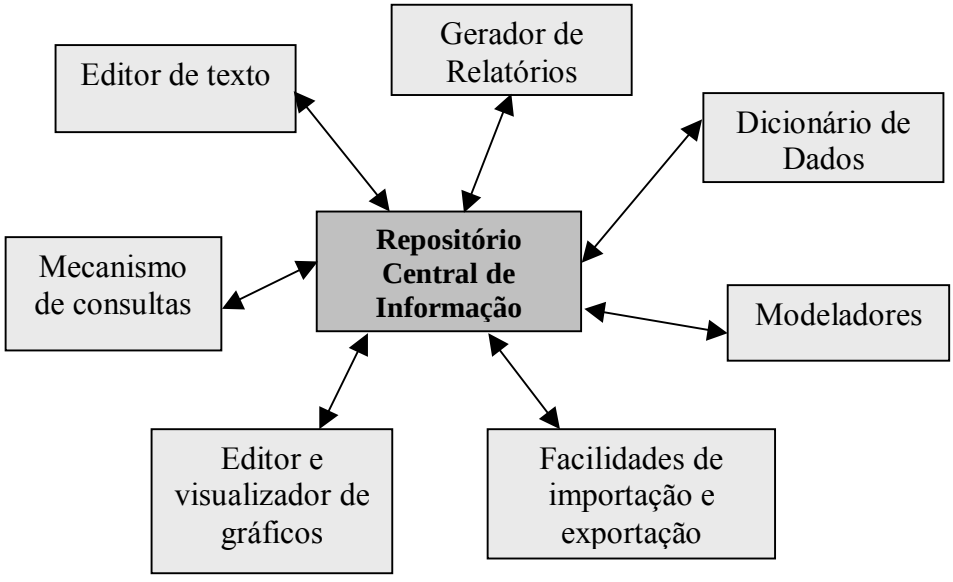

Figura 24 - Arquitetura do *ONTOCADE*

O conjunto de ferramentas para modelagem de domínio e de projeto é formado por dez componentes:

a) **Modelador de Objetivos**. Responsável pela criação do Modelo de Objetivos;

b) **Modelador de Papéis**. Responsável pela criação do Modelo de Papéis;

c) **Modelador de Pacotes**. Cria o Modelo de Pacotes;

d) **Modelador de Interações entre Papéis**. Cria o Modelo de Interações que ocorrem entre papéis;

e) **Modelador de Conceitos**. Cria o Modelo de Conceitos;

f) **Modelador de Agentes**. Responsável pela criação do Modelo de Agentes;

g) **Modelador do *Framework***. Modela o projeto do *framework*;

h) **Modelador de Atividades**. Constrói o Modelo de Atividades;

i) **Modelador de Comportamento dos Agentes**. Responsável pela criação do Modelo de Comportamento dos agentes;

j) **Modelador de Interações entre Agentes**. Cria o Modelo de Interações entre Agentes.

5.3 Automatização das atividades da *MADEM* pelo *ONTOCADE*

Esta seção abordará a modelagem do *ONTOCADE*, com o objetivo de mostrar como as ferramentas do ambiente colaboram entre si e como elas automatizam as atividades definidas pela *MADEM*.

Nesta etapa de elaboração do ambiente, será feito um projeto de alto nível para definir uma arquitetura que promova a ligação entre o *ONTOCADE* e o seu ambiente de execução. A idéia final é fazer um plano para a implementação do ambiente.

O primeiro passo para a elaboração do projeto do *ONTOCADE* é a determinação dos requisitos de utilização, ou seja, como deve ocorrer a interação do usuário com o ambiente. Para este propósito serão utilizados casos de usos com a captura dos principais cenários de interação.

A partir daí, todas as ferramentas do *ONTOCADE* serão modeladas em diagramas de classes, de interação (de seqüência e colaboração), de estado e de atividades. Em seguida, serão apresentadas as soluções de projeto.

5.3.1 Esboço do projeto

Os casos de uso a seguir capturam os passos que devem ser seguidos pelo usuário na sua interação com o sistema. Cada atividade definida pela *MADEM* será descrita por um caso de uso:

I) *Modelagem de objetivos*

Pré-condição: O usuário deve conhecer o domínio ou problema a ser modelado.
1. O usuário identifica o objetivo geral.
2. O usuário refina o objetivo geral.
3. O usuário identifica os objetivos específicos.
4. Para cada objetivo específico, o usuário identifica as responsabilidades associadas.
5. O sistema gera o modelo de objetivos.

II) *Modelagem de papéis*

Pré-condição: Pelo menos uma responsabilidade deve ser conhecida.
1. A cada responsabilidade, o sistema associa um papel.
2. O usuário pode renomear o papel gerado pelo sistema.
3. Para cada responsabilidade, o usuário define as atividades para exercê-la.
4. O usuário identifica as entradas, as saídas, os estados e os recursos para a realização das atividades.
5. O sistema gera o modelo de papel.
Alternativa: Vários papéis executam uma atividade.
No item 3, pode ocorrer que uma atividade ou uma série de atividades relacionadas é executada por vários papéis. Neste caso, o usuário cria um papel independente.

III) *Modelagem de variabilidade*

Pré-condição: Os modelos de objetivos e de papéis devem ser conhecidos.
1. O usuário classifica as instâncias dos conceitos em fixos ou variáveis.

IV) *Modelagem de Pacotes*

Pré-condição: Pelo menos um papel variável deve ser conhecido.
1. Para cada objetivo específico, o sistema gera um pacote contendo os papéis variáveis que auxiliam no cumprimento daquele objetivo específico.
2. O usuário pode renomear o pacote.

V) *Modelagem de interações entre papéis*

Pré-condição: Os objetivos específicos devem ser conhecidos.
1. Para cada objetivo específico, o sistema gera um modelo incompleto de interações entre os papéis associados àquele objetivo específico.
2. O usuário completa cada modelo gerado, identificando, seqüencialmente, as interações entre papéis e entidades externas.

VI) *Modelagem de conceitos*

Pré-condições: Análise de fontes de informação e consultas a especialistas.
1. O usuário analisa fontes de informação específicas.
2. O usuário consulta especialistas.
3. O usuário identifica os conceitos relevantes do domínio.
4. O usuário analisa aplicações existentes no referido domínio para a coleta de pontos comuns a elas.
5. O usuário completa os padrões de análise existentes ou combina-os a outros para compor um modelo de conceitos.
6. O sistema gera o modelo de conceitos.

VII) *Modelagem de agentes*

Pré-condição: Os papéis devem ser identificados.

1. O usuário identifica cada agente, selecionando os papéis que irão compô-lo.
2. O sistema gera o modelo do agente identificado.

VIII) *Modelagem do* framework

Pré-condição: Os agentes devem ser identificados.

1. O usuário organiza os agentes, seqüencialmente, de forma a colaborarem entre si.
2. O usuário analisa o problema e identifica características determinantes.
3. O usuário compara estas características com os padrões disponíveis.
4. O usuário seleciona um ou mais padrões.
5. O usuário aplica ao esboço os mecanismos de coordenação e cooperação definidos pelo padrão selecionado.
6. O sistema gera o modelo de projeto do framework.

IX) *Modelagem de atividades*

Pré-condição: O modelo de agentes deve estar construído.

1. A partir do modelo de agentes, o sistema gera o modelo de atividades.
2. O usuário pode redefinir o modelo.

X) *Refinamento dos agentes*

Pré-condição: Consulta a um sistema de padrões.

1. O usuário analisa o problema que o agente pretende resolver.
2. O usuário seleciona um ou mais padrões correlatos ao problema.
3. O usuário analisa as forças do padrão selecionado.
4. O usuário classifica os agentes em reativos ou deliberativos.

XI) *Modelagem de comportamentos*

Pré-condição: O modelo de atividades deve estar construído.
1. O sistema identifica o comportamento de cada agente a partir do modelo de atividades.
2. O usuário refina os comportamentos nos casos onde se aplicam relações do tipo "1..n para 1..n".
3. O sistema gera o modelo de comportamento do agente.

XII) *Modelagem de interações entre agentes*

Pré-condição: Os agentes devem ser identificados.
1. O sistema gera o modelo de interações entre agentes a partir do modelo de interações entre papéis suprimindo algumas interações.
2. O usuário refina o diagrama com as interações existentes entre agentes e entidades externas.

As seguintes figuras mostram, respectivamente, o Diagrama de Classes do *ONTOCADE*, o Diagrama de Colaboração das ferramentas do *ONTOCADE*, Diagramas de Atividades dos modeladores, Diagramas de Seqüência e Diagramas de Estados. Estes diagramas foram modelados com a ferramenta *Rational Rose* [3].

A Figura 25 mostra o Diagrama de classes do *ONTOCADE*, onde cada classe representa um modelador, um modelo ou um conceito. As classes específicas para cada modelador do ambiente herdam a partir da classe genérica "Modelador". Um modelador gera um ou mais modelos correspondentes. Cada modelo agrega conceitos particulares. Por exemplo, a classe "Modelador de Objetivos" gera um ou mais "Modelos de Objetivos", que por sua vez agrega os conceitos "Objetivo Geral", "Objetivos Específicos" e "Responsabilidades".

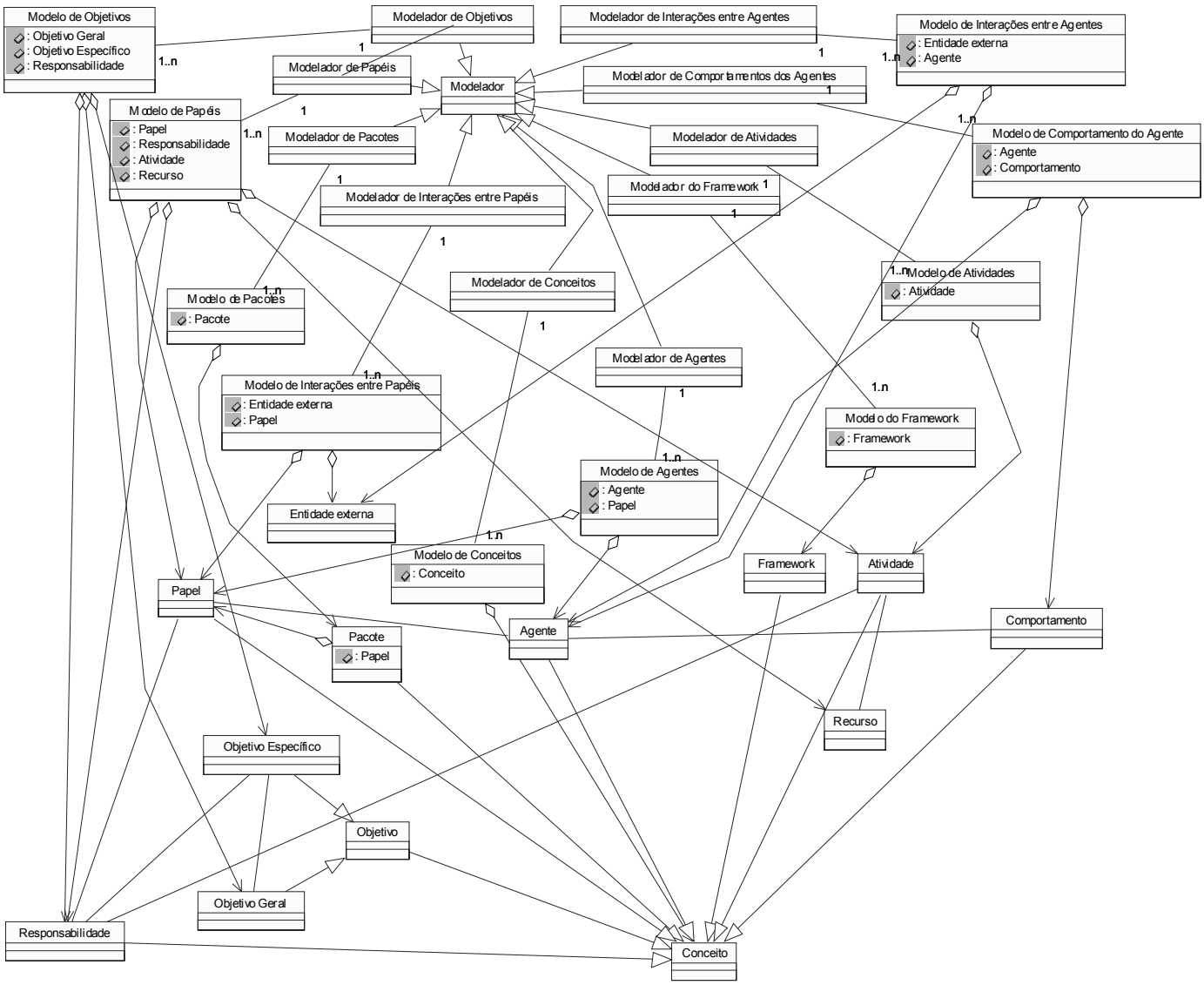

Figura 25 - Diagramas de Classes do *ONTOCADE*

A Figura 26 mostra o Diagrama de colaboração entre os modeladores do *ONTOCADE* durante todo o processo de modelagem, da análise ao projeto de domínio. Este diagrama mostra claramente como os modeladores estão ligados entre si. As mensagens trocadas entre eles enfatizam a dependência durante a modelagem, haja vista os conceitos e/ou modelos definidos numa atividade são pré-requisitos para a atividade seguinte.

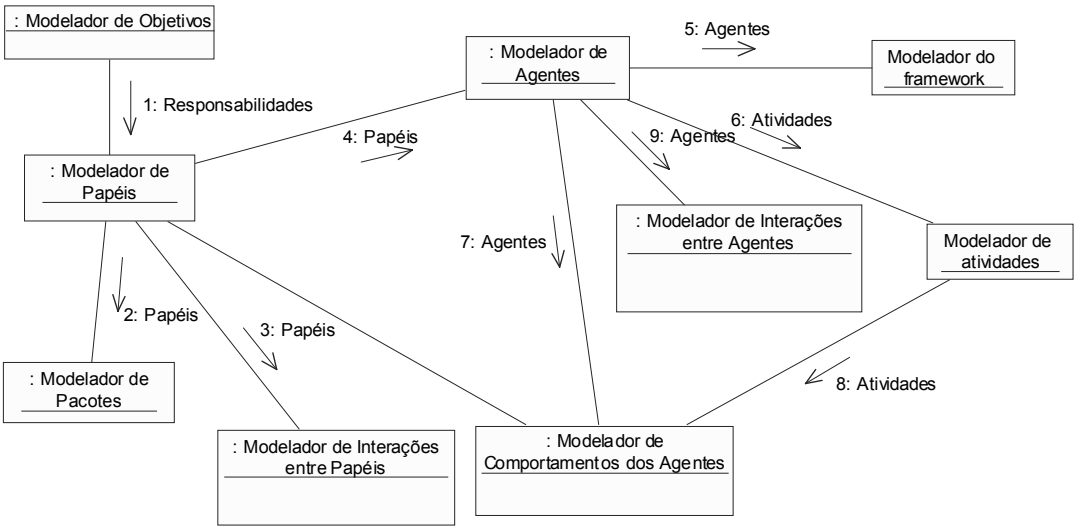

Figura 26 - Diagrama de Colaboração das ferramentas do *ONTOCADE*

As Figuras 29 e 30 mostram os Diagramas de atividades das fases de análise e de projeto, respectivamente.

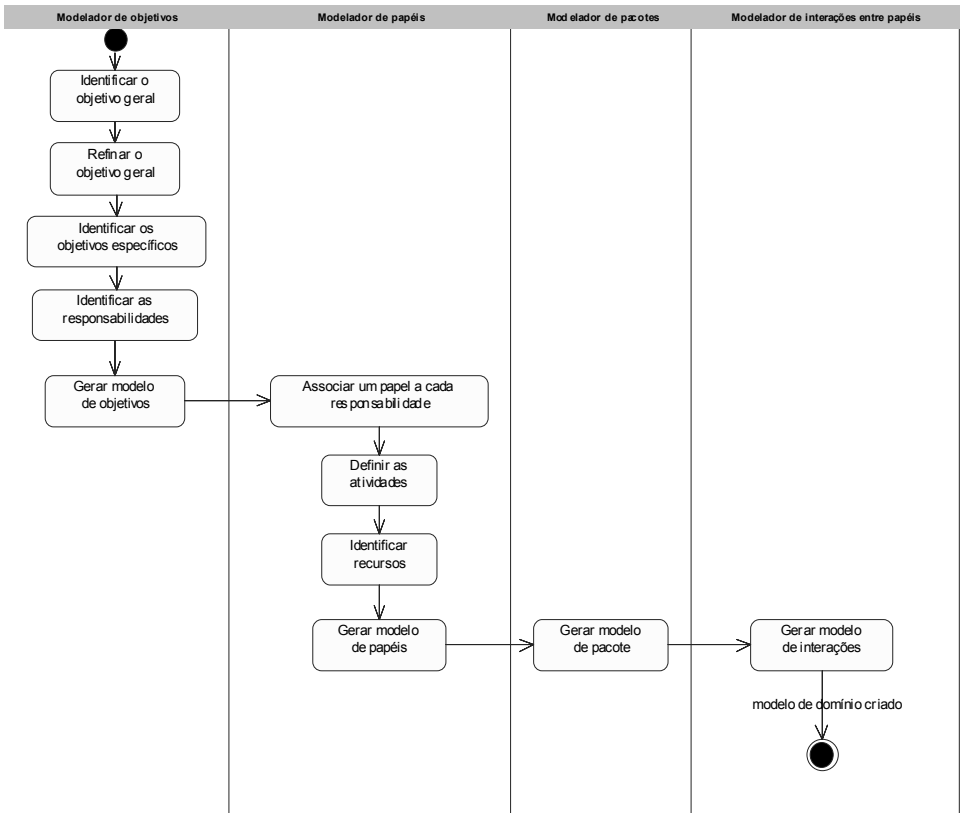

Figura 27 - Diagrama de atividades da análise de domínio

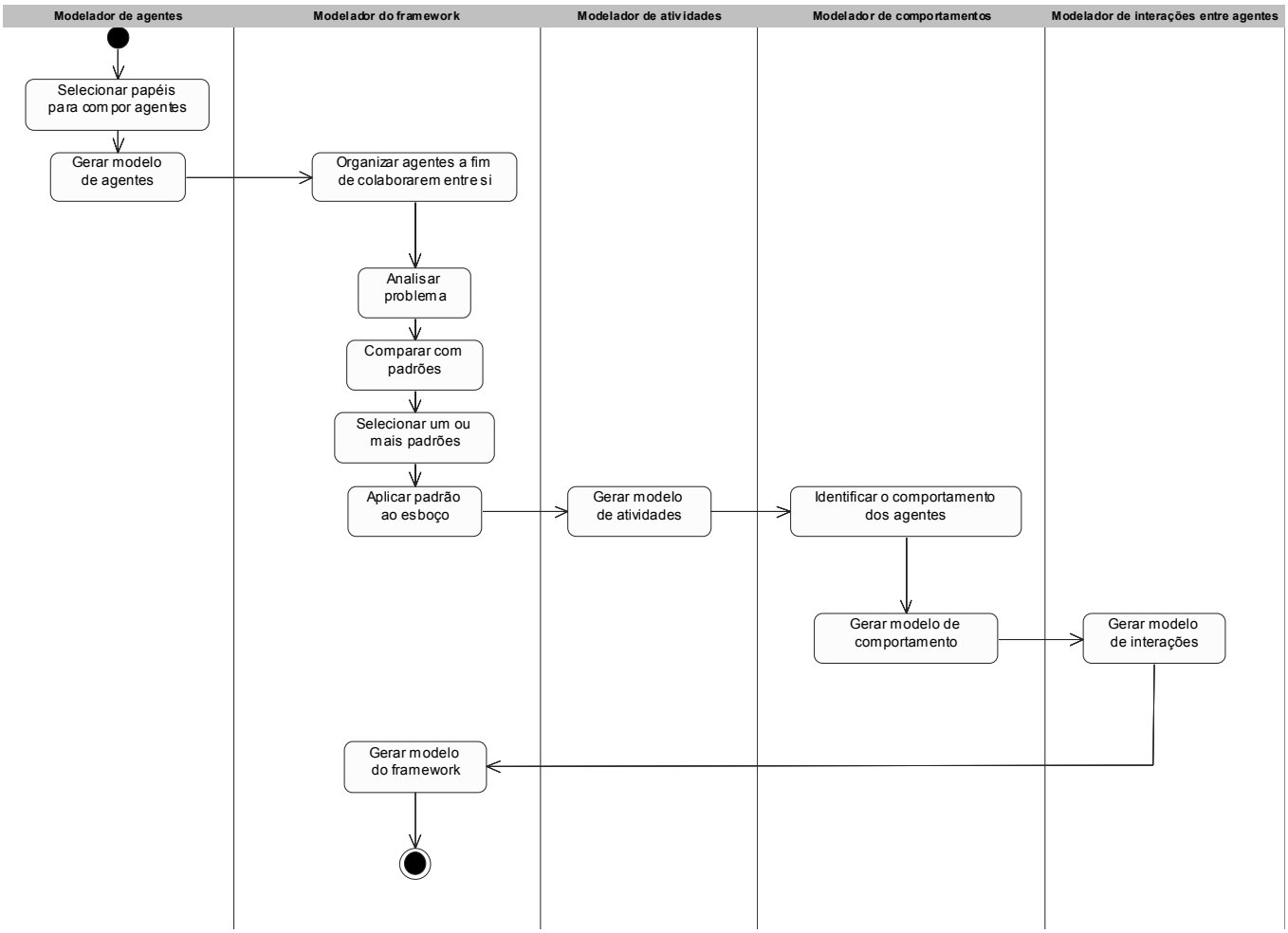

Figura 28 - Diagrama de atividades do projeto de domínio

A Figura 29 exibe o diagrama seqüencial de interações entre as ferramentas do *ONTOCADE* que são invocadas durante o processo de análise de domínio. O usuário, inicialmente, carrega o modelador de objetivos a partir da interface.

O usuário só pode passar de uma fase a outra quando os requisitos mínimos para realização de uma determinada atividade tiverem sido definidos. Ao acessar a interface, o projetista transfere para o modelador de objetivos todo seu conhecimento acerca do domínio a ser modelado. Após definir as responsabilidades, estas são passadas ao modelador de papéis que gera automaticamente os modelos de papéis. Quando os modelos de objetivos e de papéis estão concluídos, é feita a modelagem de variabilidade, quando, então, é possível a criação dos modelos de pacotes assim que os papéis variáveis forem identificados.

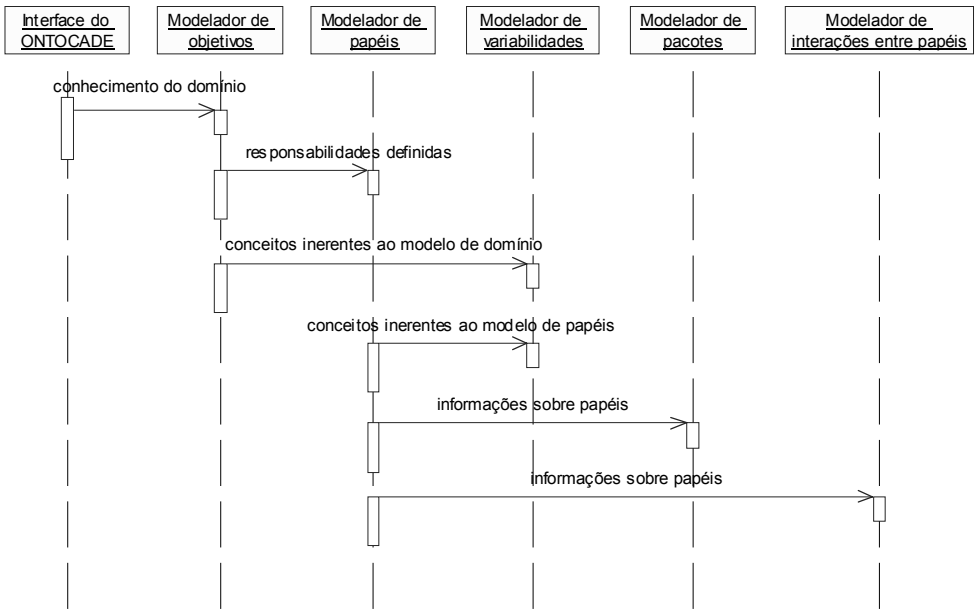

Figura 29 - Diagrama de seqüência da análise de domínio

A Figura 30 mostra o diagrama seqüencial de interações entre as ferramentas do ambiente durante o processo de projeto de domínio. O ambiente utiliza nesta fase todas as informações existentes no modelo de domínio criado na fase anterior de análise.

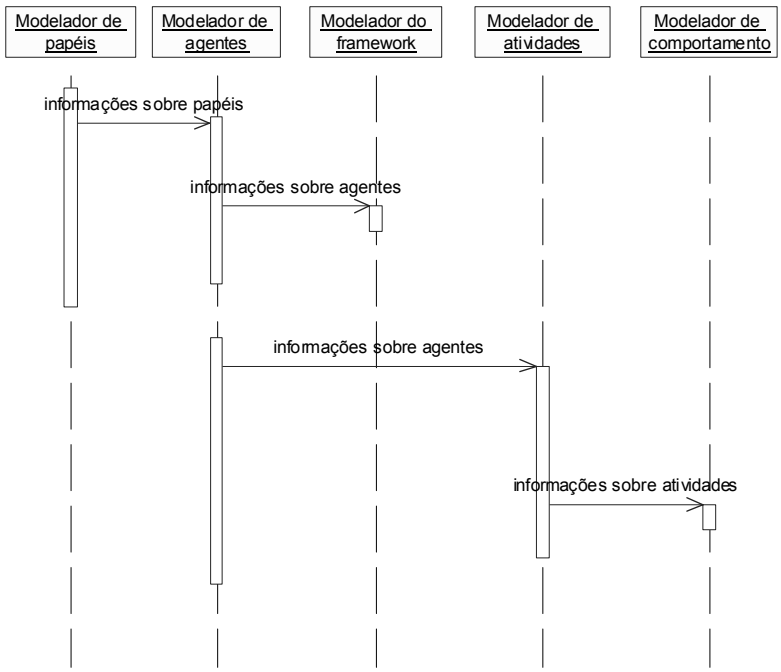

Figura 30 - Diagrama de seqüência do projeto de domínio

As Figuras 33 a 35 mostram os diagramas de estado das ferramentas que compõem o *ONTOCADE* das fases de análise e de projeto de domínio.

Na Figura 31, quando existe uma nova formulação de um problema a ser modelada, são realizadas apenas as modelagens de objetivos, de papéis, de variabilidades, de pacotes e de interações entre papéis.

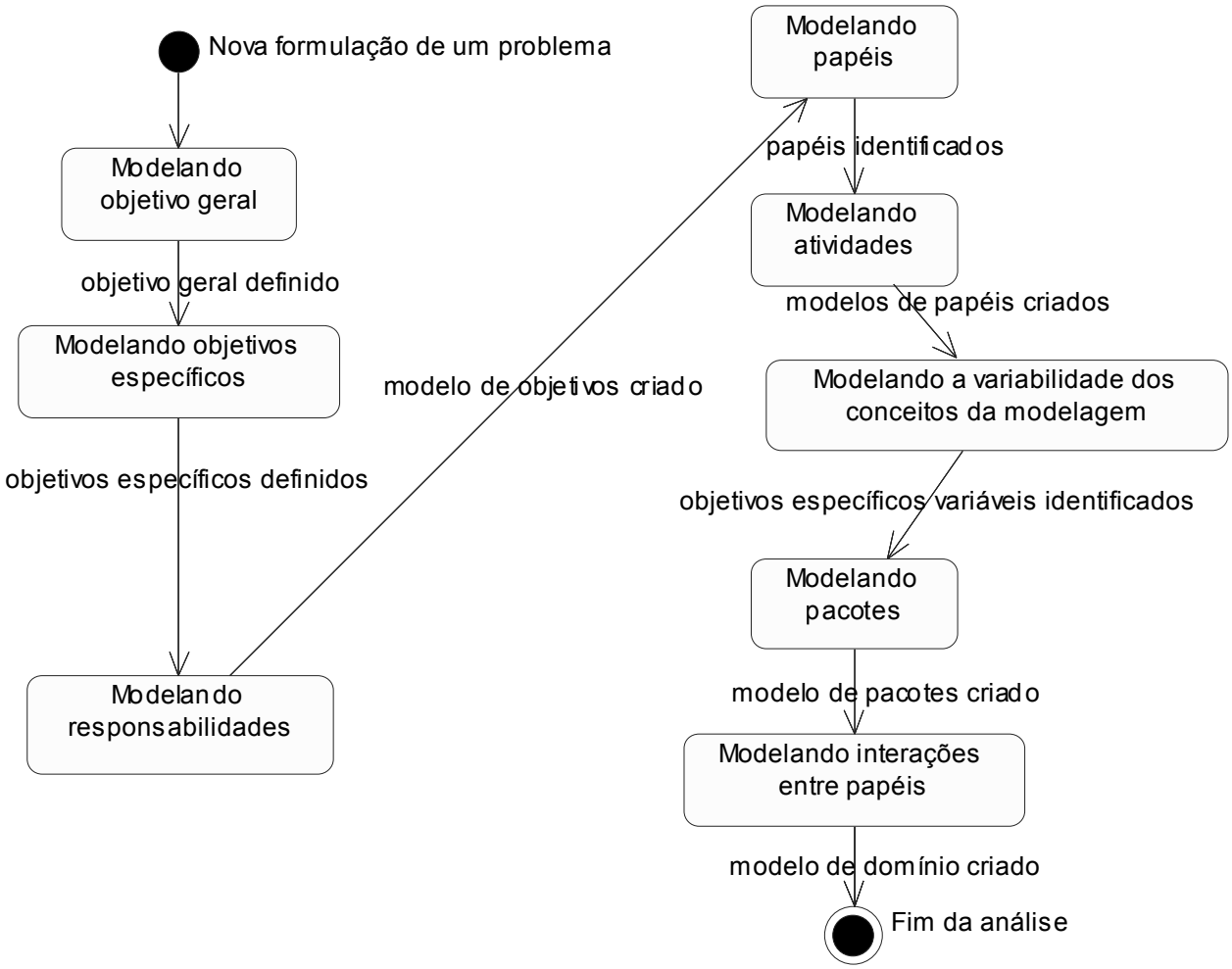

Figura 31 - Diagrama de Estados da modelagem de um problema

Na Figura 34, quando há uma nova área de conhecimento a ser modelada, apenas as modelagens de conceitos e de variabilidades são realizadas.

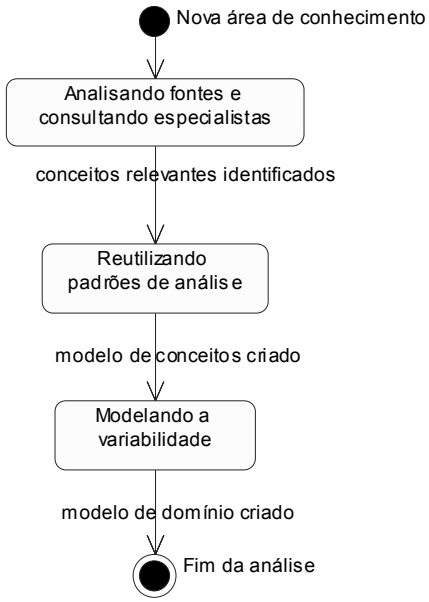

Figura 32 - Diagrama de Estados da modelagem de uma área de conhecimento

Na Figura 35, quando há um novo modelo de domínio, inicia-se a modelagem de projeto, que culmina na construção de um *framework*.

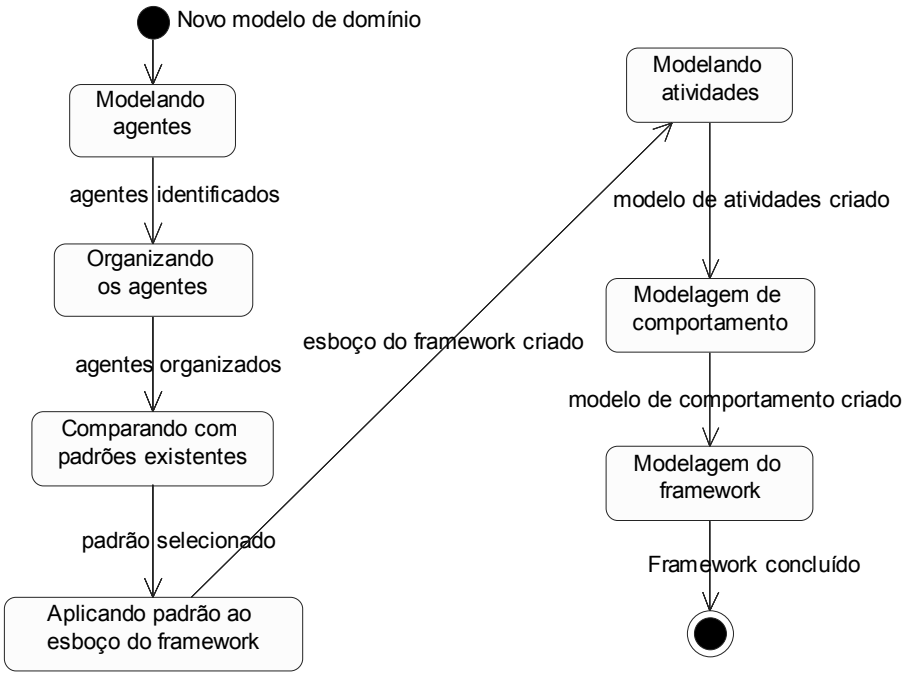

Figura 33 - Diagrama de Estados da fase de projeto de domínio

Como alternativa à UML, é conveniente utilizar, também, a própria MADEM na modelagem do ONTOCADE (ver Anexo).

5.3.2 Soluções de projeto

O *ONTOCADE* é escrito na linguagem de programação Java [16] [34], caracterizando-se, portanto, como um ambiente portável e interoperável. Como ambiente de execução foi escolhido o ambiente *Protégé* [55].

A escolha do *Protégé* deve-se aos seguintes fatores:

a) Trata-se de uma ferramenta de código-aberto;

b) É escrita na linguagem *Java*;

c) Destina-se à modelagem de ontologias e aquisição de conhecimento;

d) É um ambiente extensível;

e) Oferece serviços que podem ser usufruídos pelas extensões, tais como mecanismos de armazenamento de dados, bibliotecas de componentes adaptáveis, mecanismos para tratamento de eventos, dentre outros;

f) É amplamente utilizada por usuários e projetistas na área de sistemas de bases de conhecimento;

As extensões ao *Protégé* são conhecidas por *plugins*. Existem três tipos de *plugins*: *tab-widget*, *slot-widget* e *backend*. *Tab-widget* é uma interface de usuário utilizada para a execução de algumas tarefas, como criação de classes e execução de consultas. *Slot-widget* é usado para visualizar e adquirir um valor para um *slot*. *Backend* é usado para especificar um mecanismo de armazenamento de dados (p. ex., arquivo-texto ou banco de dados).

Para a execução do *ONTOCADE*, o *Protégé* foi configurado para acomodar um *plugin* do tipo *tab-widget* e uma extensão ao metamodelo. O metamodelo do *Protégé* será estendido para incorporar conceitos e regras definidos pela *MADEM*; os modeladores serão acessados a partir deste *plugin*.

A Figura 34 mostra a arquitetura detalhada do ambiente *ONTOCADE*.

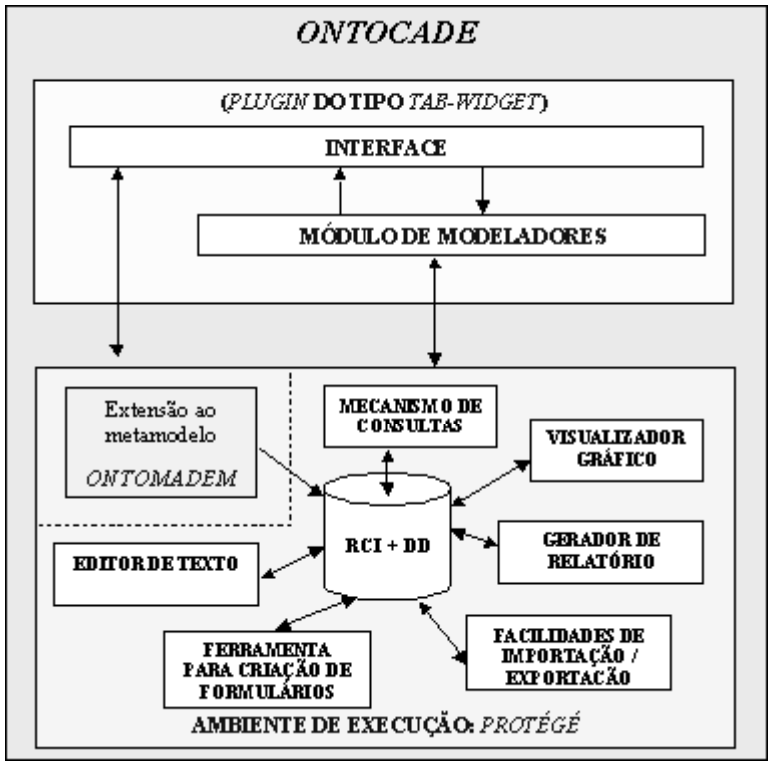

Figura 34 - Arquitetura detalhada do *ONTOCADE*

De acordo com a Figura 34, o *ONTOCADE* executa no ambiente *Protégé*, que possui ferramentas para a realização de tarefas de propósito geral. O *plugin* dispõe das ferramentas específicas para análise e projeto de domínio multiagente e uma interface para acessar todas as ferramentas, genéricas e específicas.

A extensão ao metamodelo do *Protégé* foi feita com a reutilização da ontologia genérica *ONTOMADEM* [20], que é utilizada para guiar a aplicação das regras da *MADEM*. As ferramentas providas pelo *Protégé* foram reutilizadas pelo *ONTOCADE*.

O RCI e o DD são providos pelo próprio mecanismo de armazenamento e gerenciamento de dados do *Protégé*. Da mesma forma, são utilizados os mecanismos existentes no *Protégé* para edição de textos, geração de relatórios, manipulação de consultas, criação de formulários, importação/exportação e edição/visualização de gráficos.

O Editor de textos permite a inserção de notas ou comentários que não fazem parte da estrutura da ontologia, mas fornecem informações adicionais sobre classes, instâncias e *slots*. A Figura 35 mostra um exemplo de nota associada a uma classe da ontologia.

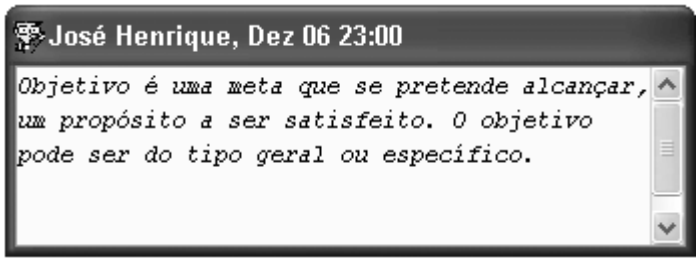

Figura 35 - Exemplo de comentário editado pelo Editor de Notas

O Gerador de relatório reporta o projeto de ontologia num arquivo *HTML* (*Hypertext Markup Language*), que permite a visualização da hierarquia de classes e todas as instâncias. O relatório consiste de uma página de índices, que fornece a hierarquia de classes e *links* para páginas individuais sobre cada classe. Estas páginas incluem descrições de *slots* e instâncias. As instâncias que forem selecionadas para a geração de relatórios, são exibidas em páginas individuais e aparecem no índice hierárquico e abaixo de cada classe. A Figura 36 mostra o trecho de um índice gerado pelo Gerador de relatórios.

Class Hierarchy for *ONTOINFO-MADEM* Project

- Conceitos da Modelagem
  - Objetivo
    - Objetivo Geral
      Instances : Satisfazer necessidades de informacao
    - Objetivo Especifico
      Instances : Atender necessidades pontuais explicitas, Atender necessidades duradouras explicitas, Atender necessidades pontuais implicitas, Atender necessidades duradouras implicitas
  - Responsabilidade
    Instances : Mineracao de uso, Mineracao de dados, Modelagem de Usuario, Interacao com usuario, Estruturacao na fonte de informacao, Selecao de informacao, Monitoramento da fonte de informacao
  - Papel Interno
    Instances : Estruturador, Interfaceador, Seletor, Minerador de Dados, Interfaceador, Monitor das Fontes de Informacao, Seletor, Modelador de Usuário , Minerador de Uso

Figura 36 - Trecho de um índice criado pelo gerador de relatórios

O primeiro passo adotado na construção do *ONTOCADE* foi a incorporação da *ONTOMADEM* ao *Protégé*, como extensão ao metamodelo. Este metamodelo é, ele próprio, uma ontologia que representa a hierarquia de classes genéricas com raiz em *:THING*. Todas as classes, inclusive as específicas de domínio, descendem, direta ou indiretamente, da classe *:THING*.

O *Protégé* possui um painel que permite a adição de classes ao metamodelo. Todos os conceitos de modelagem definidos pela metodologia *MADEM* foram inseridos na

hierarquia como metaclasses que poderão ser instanciadas para a definição de modelos de domínio e de projeto. É exatamente esta hierarquia de metaclasses que representa a ontologia genérica *ONTOMADEM*, que servirá de guia para a execução da metodologia *MADEM*.

Toda metaclasse definida no *Protégé* herda de *:METACLASS*, uma classe genérica sob *:THING*. Na *ONTOMADEM*, três classes herdam diretamente da classe *:METACLASS*. São elas: "Conceitos da Modelagem", "Tarefas de Modelagem" e "Produtos de Modelagem". Todas estas classes são definidas como abstratas, ou seja, classes que não podem ter instâncias. A classe "Conceitos de Modelagem" é superclasse das seguintes subclasses: "Objetivo", "Responsabilidade", "Papel interno", "Atividade", "Recurso", "Dado", "Estado", "Pacote", "Papel externo", "Conceito do Domínio", "Agente", "Comportamento" e "Padrão". A Tabela 8 mostra a hierarquia de classes com raiz em "Conceitos de Modelagem", com seus respectivos tipos e *slots*.

CLASSE	**SUPERCLASSE**	**TIPO**	***SLOTS* DIRETOS**
Conceitos da Modelagem	:THING	abstrata	nome descrição
Objetivo	Conceitos da Modelagem	abstrata	variabilidade
Objetivo Geral	Objetivo	concreta	-
Objetivo Específico	Objetivo	concreta	-
Responsabilidade	Conceitos da Modelagem	concreta	variabilidade
Papel interno	Conceitos da Modelagem	concreta	variabilidade
Atividade	Conceitos da Modelagem	concreta	variabilidade pos-condições pré-condições entrada saída recurso
Recurso	Conceitos da Modelagem	concreta	variabilidade
Dado	Conceitos da Modelagem	abstrata	-
Dado de entrada	Dado	concreta	-
Dado de saída	Dado	concreta	-
Estado	Conceitos da Modelagem	abstrata	-
Estado anterior	Estado	concreta	pre-condicao de
Estado posterior	Estado	concreta	pos-condicao de
Pacote	Conceitos da Modelagem	concreta	composto por
Papel externo	Conceitos da Modelagem	concreta	variabilidade
Conceito do domínio	Conceitos da Modelagem	concreta	-
Agente	Conceitos da Modelagem	concreta	-
Comportamento	Conceitos da Modelagem	concreta	entrada
Padrão	Conceitos da Modelagem	abstrata	saída
Padrão de análise	Padrão de análise	concreta	-
Padrão arquitetural	Padrão de análise	concreta	-
Padrão de *projeto detalhado*	Padrão de análise	concreta	-

Tabela 8 - Superclasse "Conceitos de Modelagem" e suas subclasses

A classe abstrata “Tarefas de Modelagem” é superclasse das seguintes subclasses concretas: “Modelagem de Domínio”, “Modelagem de Objetivos”, “Modelagem de Papéis”, “Modelagem de Variabilidade”, “Modelagem de Pacotes”, “Modelagem de Interações entre Papéis”, “Modelagem de Conceitos”, “Projeto de Domínio”, “Modelagem de Agentes”, “Modelagem do *Framework*”, “Modelagem de Atividades”, “Refinamento dos Agentes”, “Modelagem do Comportamento” e “Modelagem de Interações entre Agentes”.

A classe abstrata “Produtos de Modelagem” é superclasse das seguintes subclasses concretas: “Modelo de Domínio”, “Modelo de Objetivos”, “Modelo de Papéis”, “Modelo de Pacotes”, “Modelo de Interações entre Papéis”, “Modelo de Conceitos”, “*Framework*”, “Modelo de Agentes”, “Modelo de Projeto do *Framework*”, “Modelo de Atividades”, “Modelo de Comportamento” e “Modelo de Interações entre Agentes”.

5.4 Implementação do *plugin* e da interface

Esta seção apresenta a programação do código-fonte (ver Apêndice) do *plugin* e da interface do *ONTOCADE*. Neste trabalho não serão abordadas as implementações dos modeladores, mas apenas as bases para integração das mesmas ao ambiente. A programação ocorreu conforme os seguintes passos:

a) Desenvolvimento de um código-esqueleto que será a base para o código específico do *ONTOCADE*;

b) Integração do esqueleto aos códigos específicos das ferramentas do sistema;

c) Definição de um diretório de empacotamento das classes, utilizando a convenção de domínio invertido. Neste caso, o pacote definido para armazenar os arquivos binários é “ontocade.br.ufma.deinf.maae”, rótulo composto pelo nome do ambiente proposto e pelo domínio invertido do projeto *MaAE* [29];

d) Compilação os códigos-fonte para a geração dos arquivos *.class*;

e) Colocação do pacote “ontocade.br.ufma.deinf.maae” dentro da pasta “plugin”, localizada no diretório no qual foi instalado o *Protégé*;

f) Criação de um arquivo “manifest” com o caminho do arquivo *.class* gerado após a compilação dos códigos-fonte.

A Figura 37 mostra o código do esqueleto para a criação de qualquer *plugin* do tipo *tab-widget*. Em negrito, estão as partes que precisam ser modificadas para atender aos requisitos específicos de cada aplicação.

```
package nome do diretório de empacotamento;

import java.?;
import edu.stanford.smi.protege.?;
import ?

public class Nome_do_Plugin extends AbstractTabWidget {

    public void initialize() {    //inicia plugin
        setLabel("rótulo do plugin");

        operações específicas;

    }

    public static void main(String[] args) {
        edu.stanford.smi.protege.Application.main(args);
    }
}
```

Figura 37 - Esqueleto de um *plugin* do tipo *tab-widget* [55]

De acordo com a Figura 37, duas ações precisam ser feitas quando da adaptação do esqueleto:

a) Tornar a classe pública *Nome_do_Plugin* subclasse de *AbstractTabWidget*, uma classe que faz parte da biblioteca do *Protégé*;

b) Implementar o método *initialize ( )*, acrescentando as operações que irão ativar as classes específicas de cada ferramenta do *ONTOCADE*.

Após a execução dos passos para a construção do *plugin*, ele estará pronto para executar corretamente na plataforma do *Protégé*. Quando for iniciado, o *Protégé* precisa ser configurado para exibir o *plugin* por meio da opção “Configure...” do menu “Project”. Aparecerá uma caixa de diálogo que permite selecionar de uma lista de *plugins*, aqueles que se deseja exibir (Figura 38).

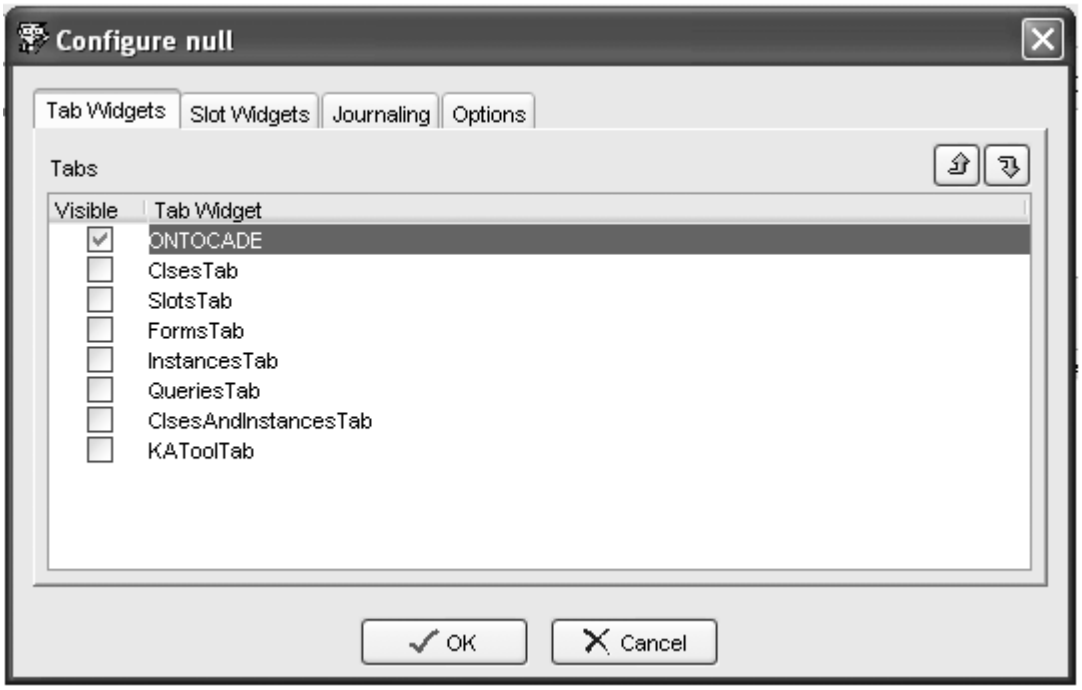

Figura 38 - Configuração de *plugins*

As setas (⇧⇩) disponíveis na caixa de configuração permitem a alteração da ordem de exibição dos *plugins* disponíveis. Quando um projeto (*.pprj*) é salvo, esta ordem é mantida para ele, não havendo necessidade, portanto, de nova configuração.

Outra forma possível de armazenar os arquivos do ambiente é em um arquivo .jar executável.

5.5 Diretrizes de utilização do *plugin*

A execução do *ONTOCADE* inicia clicando-se no arquivo *ontocade.pprj*, que aciona o *Protégé*, a *ONTOMADEM* e o *plugin* a partir do qual os modeladores podem ser acessados. Como mostra a Figura 39, a interface usuário-ambiente permite, por meio dos botões, o acesso a todos os modeladores do *ONTOCADE*.

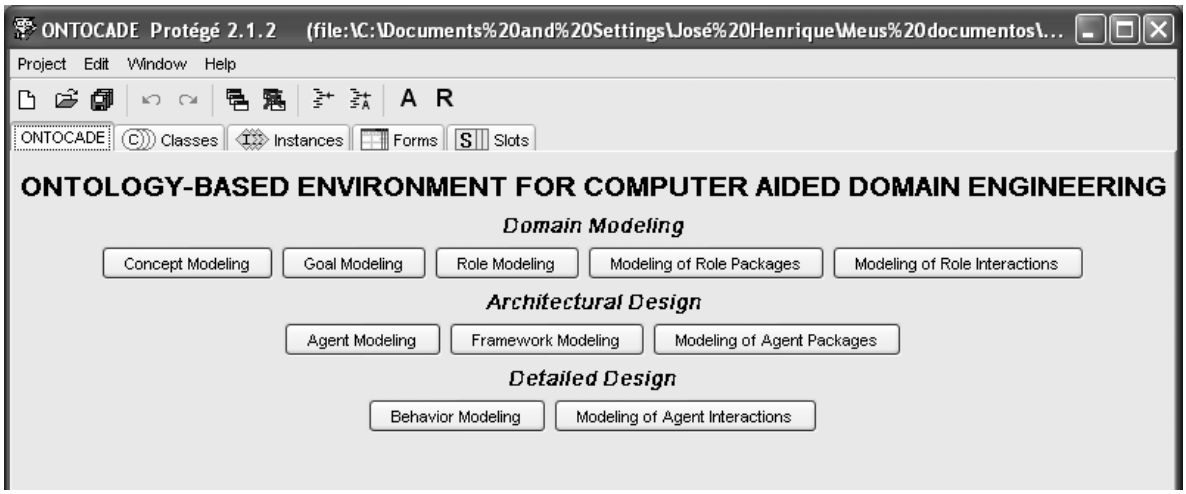

Figura 39 - Interface do *ONTOCADE*

A primeira ação a ser realizada é a criação do Modelo de Objetivos. Para isto, o usuário deve acessar o botão correspondente à Modelagem de Objetivos (*Goal Modeling*). Aparecerá então o Modelador de Objetivos (*Goal Modeler*), mostrado na Figura 40.

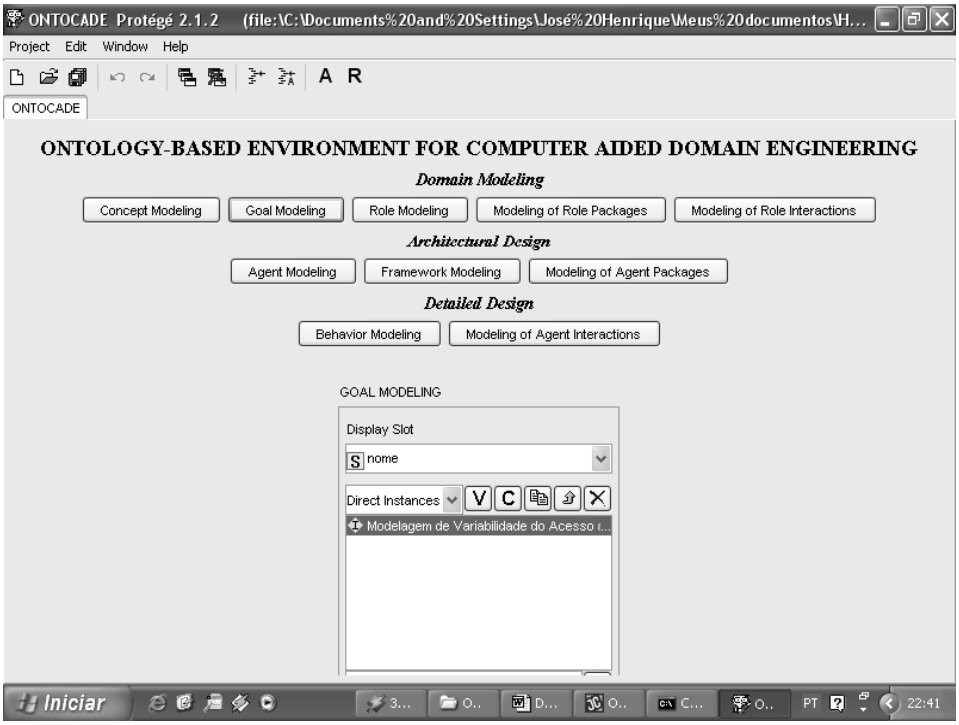

Figura 40 - Modelador de Objetivos do *ONTOCADE*

As interfaces dos modeladores possuem os mesmos dispositivos:

a) ***Display Slot***;

b) ***Direct Instances***;

c) **Botão de Visualização V (*View Button*)**. Permite a visualização dos gráficos de um modelo. É possível a visualização cliclando-se duas vezes no nome do modelo;

d) **Botão de Criação C (*Create Button*)**. Permite a criação de um novo modelo;

e) **Botão de Cópia (*Copy Button*)**. Permite "clonar" um modelo;

f) **Botão de Referências (*References Button*)**. Mostra todos os *frames* que estão diretamente relacionados com o modelo selecionado;

g) **Botão de Exclusão ✖ (*Delete Button*)**. Permite a exclusão de um modelo;

h) **Lista de modelos**. Campo que exibe todos os modelos criados;

i) **Botão Buscador (*Finder Button*)**. Permite a seleção rápida de um modelo da lista.

Quando o botão C é clicado, aparecerá no campo de modelos um nome aleatório, criado pelo *Protégé*. Clicando-se duas vezes nele ou pressionando o botão V, será mostrado editor gráfico do modelador em uso (Figura 41).

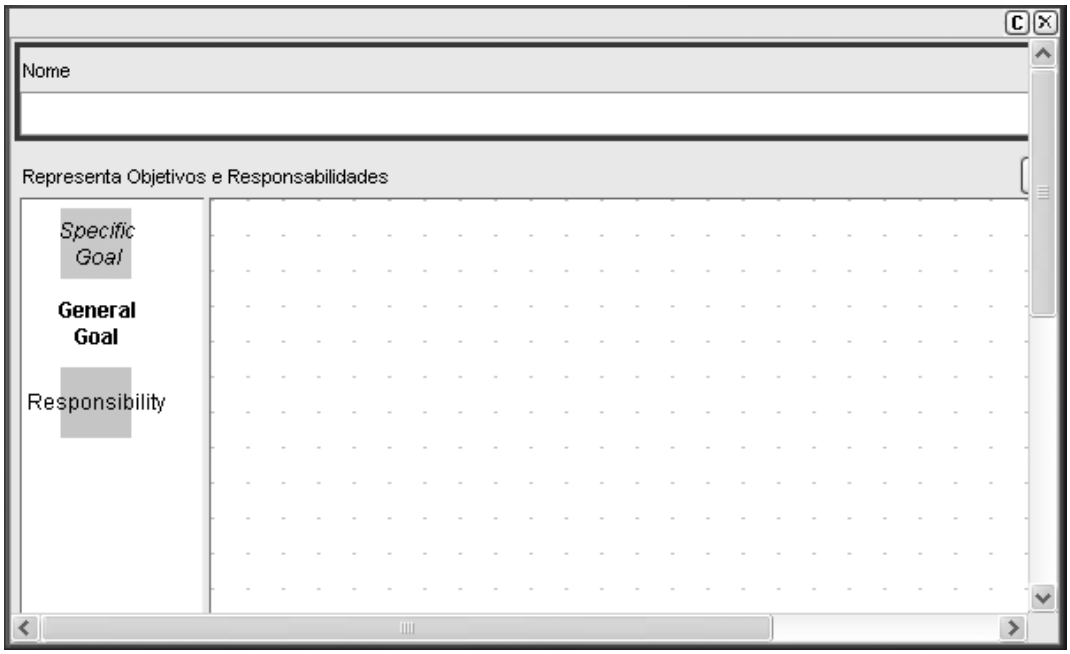

Figura 41 - Editor Gráfico do Modelador de Objetivos

No Editor Gráfico é possível renomear o modelo e construí-lo de acordo com as regras definidas pela *MADEM*. No lado esquerdo do editor aparecem apenas os conceitos relacionados ao modelo que se está construindo. Como mostra a Figura 41, para o modelo de objetivo são necessários apenas os conceitos *General Goal*, *Specific Goal* e *Responsibility*.

Os conceitos que aparecem no lado esquerdo podem ser arrastados para o lado direito, onde são instanciados para a definição dos conceitos específicos para cada domínio. Para renomear as instâncias, basta clicar duas vezes no objeto arrastado. Os nomes atribuídos farão parte, automaticamente, da hierarquia de classes do *Protégé* (Figura 44).

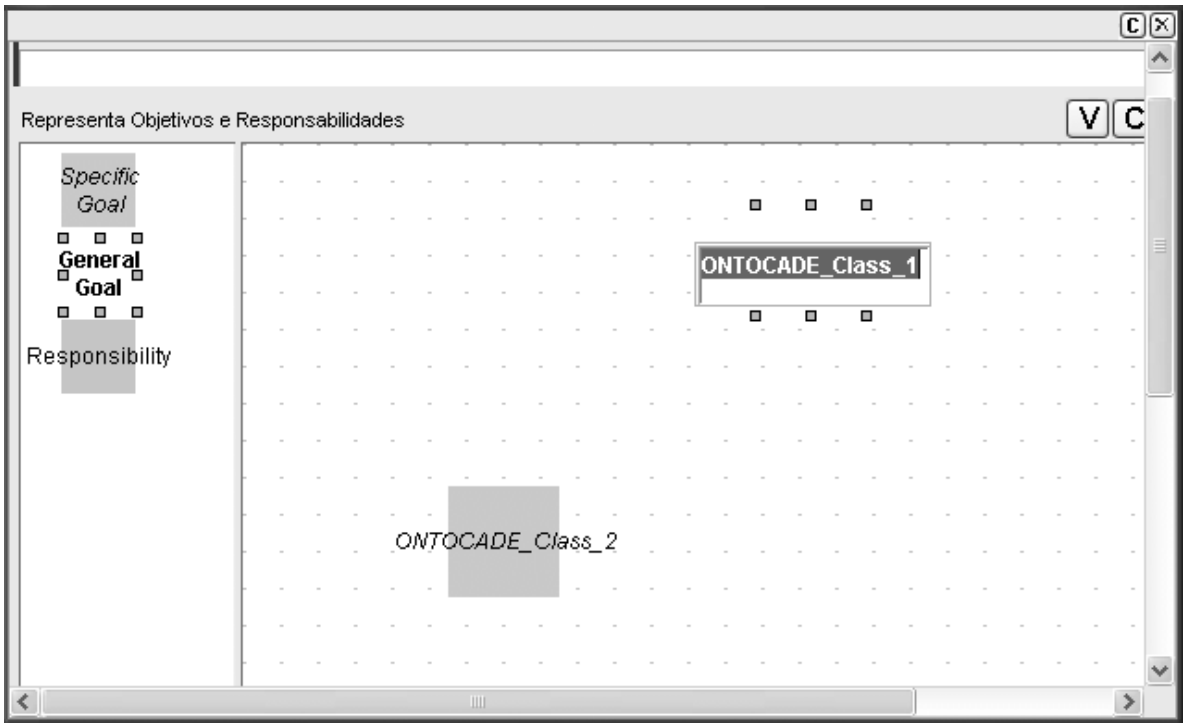

Figura 42 - Processo de instanciação de conceitos

O processo de criação dos demais modelos é semelhante ao processo de construção do Modelo de Objetivos. O usuário deve apenas clicar no botão da modelagem que se pretende realizar e o *ONTOCADE* exibe o modelador correspondente, que por sua vez, mostrará apenas os conceitos relacionados.

O *ONTOCADE* mantém controle rigoroso da execução da *MADEM*, considerando a ordem das atividades e a dependência entre as fases. Por exemplo, a Modelagem de Papéis só pode ser iniciada se pelo menos uma responsabilidade tiver sido definida na Modelagem de

Objetivos. Qualquer tentativa de violação das regras é recusada pelo *ONTOCADE*, que emite uma mensagem de alerta.

A tabela a seguir compara o *ONTOCADE* com as ferramentas revisadas na Seção 3.4. O *ONTOCADE* teve sua concepção inspirada nas ferramentas *JAFMAS, PTK* e *Agent Factory*. Em comum, automatizam as fases de análise e projeto, utilizam ontologias e oferecem suporte a métodos e às atividades de modelagem, definição de comportamento e reuso. *ONTOCADE*, diferentemente, fornece suporte à construção de produtos reutilizáveis de um processo de Engenharia de Domínio Multiagente, guiando a criação e atualização dos modelos dos sistemas, de acordo com regras definidas pela *MADEM*.

FASES DO CVD*S*		**FERRAMENTAS**						
		JADE	*MAST*	*Agent Tool*	*JAFMAS*	*PTK*	*Agent Factory*	*ONTOCADE*
Análise			•	•	•	•		•
Projeto			•	•	•	•		•
Implementação				•	•	•	•	
ATIVIDADES / CARACTERÍSTICAS	Geração de código		•	•		•	•	
	Modelagem e simulação	•			•	•		•
	Comunicação e cooperação		•	•	•	•		
	Definição de comportamento			•	•	•	•	•
	Desenvolvimento para reuso							•
	Desenvolvimento com reuso		•	•		•	•	
	Processamento de linguagem	•	•			•	•	
	Suporte a métodos			•	•	•	•	•
	Uso de ontologias		•			•		•
	Diagramação					•	•	•
	Prototipação						•	
	Documentação textual					•	•	
	Engenharia Reversa					•		

Tabela 9 - Comparação entre *ONTOCADE* e outras ferramentas baseadas em agentes

Devido a algumas limitações experimentadas na adaptação do código do *Protégé* para receber o *ONTOCADE*, alguns requisitos apontados na Seção 5.1 não foram totalmente

satisfeitos, como controle rigoroso da *MADEM* e facilitação do reuso de padrões existentes, ou seja, no estágio atual, a automatização das fases da *MADEM* oferecida pelo *ONTOCADE*, é parcial.

5.6 Considerações Finais

Este capítulo introduziu o *ONTOCADE*, um ambiente baseado em ontologias para análise e projeto de domínio multiagente. Como fornece suporte às fases iniciais do processo de desenvolvimento de *software*, este ambiente trata-se de um *CASE workbench*.

A Seção 5.1 mostrou os requisitos que o ambiente precisa atender para fornecer suporte, de forma eficiente, às atividades da *MADEM* [20] (cap. 4). Dentre estes requisitos estão: fornecer meios para captura de requisitos, garantir controle rigoroso da aplicação das regras da *MADEM* e fornecer ferramentas para a construção dos modelos do sistema.

Em seguida foi mostrada a concepção do ambiente, com a definição de sua arquitetura. O *ONTOCADE* possui dez ferramentas específicas (os modeladores), oito ferramentas genéricas (providas pelo *Protégé*) e uma ontologia genérica, a *ONTOMADEM* [20], que representa o conhecimento da *MADEM*.

A Seção 5.3 mostrou como é feita a automatização das atividades da *MADEM* pelo *ONTOCADE*, destacando o esboço do projeto do ambiente e as soluções para o alcance dos requisitos do ambiente proposto.

A Seção 5.4 apresentou os passos da implementação das ferramentas e do *plugin* responsável pelo acesso a todos os componentes do ambiente e pela integração dos mesmos ao *Protégé*. A escolha do *Protégé* como ambiente de execução do *ONTOCADE* deveu-se, dentre outros, aos seguintes fatores: é plataforma de código aberto, escrita em Java, extensível e muito utilizado na modelagem de ontologias e aquisição de conhecimento. A Seção 5.5 mostrou as diretrizes de utilização do ambiente. Todos os modeladores possuem interfaces amigáveis, permitindo que o usuário execute, intuitivamente, todas as tarefas necessárias para a análise e projeto de domínio.

No capítulo de conclusões serão discutidas algumas das limitações atuais do ambiente proposto neste trabalho e as correspondentes alternativas para solucioná-las.

6 CONCLUSÕES

Este trabalho propôs o *ONTOCADE*, um ambiente *CASE* baseado em ontologias para análise e projeto na Engenharia de Domínio Multiagente.

ONTOCADE provê suporte à aplicação das fases da *MADEM*, uma metodologia que guia a captura e especificação de requisitos de uma família de sistemas, em um domínio de aplicação, para a geração do modelo de domínio e a captura e especificação do projeto reutilizável para a geração de um *framework* multiagente.

O ambiente é baseado na ontologia genérica *ONTOMADEM*, que visa garantir a execução correta das fases e atividades da *MADEM*. Esta ontologia é resultante do mapeamento da representação semântica dos conceitos da metodologia *MADEM* a uma hierarquia de metaclasses, que foi empregada para estender o metamodelo da plataforma *Protégé*.

Com o metamodelo estendido para conter os conceitos da *MADEM*, qualquer domínio pode ser modelado de acordo com as diretrizes impostas pela metodologia.

6.1 Resultados e Contribuições da Pesquisa

As principais atividades realizadas e resultados obtidos nesta pesquisa foram:

a) **Análise das ferramentas *CASE* existentes para o desenvolvimento de aplicações multiagentes**. Constatou-se que não existem ferramentas para a Engenharia de Domínio Multiagente, mas, em contrapartida, há um grande número de ferramentas para a Engenharia de Aplicações Multiagentes, sobretudo para o suporte às atividades de comunicação e colaboração entre agentes de uma sociedade. Portanto, este trabalho contribui de forma significativa com o desenvolvimento *para* a reutilização no paradigma de agentes;

b) **Concepção, desenvolvimento e implementação de uma ferramenta *CASE* para a Engenharia de Domínio Multiagente**. *ONTOCADE* é uma proposta de ferramenta para a modelagem e o projeto de domínio, cujo principal objetivo é produzir uma arquitetura reutilizável, ou seja, uma solução baseada em agentes para um problema especificado em um determinado domínio. O ambiente

proposto fornece suporte a *MADEM*, uma metodologia para análise e projeto na Engenharia de Domínio Multiagente, automatizando a execução de suas atividades;

6.2 Trabalhos Futuros

Como sugestões para trabalhos futuros, citam-se:

a) **Avaliação da ferramenta proposta**. Visando a certificação da efetividade do *ONTOCADE*, propõe-se a realização de um estudo de caso com o desenvolvimento de um *framework* para o desenvolvimento de sistemas para o acesso à informação, por exemplo;.

b) **Desenvolvimento de um ambiente para a Engenharia de Aplicações Multiagente**. Sugere-se a criação de uma ferramenta para o desenvolvimento de aplicações multiagentes que empregaria a representação das ontologias e modelos desenvolvidos com o *ONTOCADE*. Neste caso, esta nova ferramenta seria também um *plugin* do *Protégé* que trabalharia em conjunto com o *ONTOCADE*;

c) **Construção de *frameworks* com a utilização do *ONTOCADE***. Com o objetivo de agregar conhecimento, outros domínios de aplicação devem ser experimentados com a construção de *frameworks* nas mais diversas áreas;

d) **Extensão do *ONTOCADE* para dar suporte automatizado às atividades da *MADEM***. Neste trabalho foram implementados apenas o *plugin* do *ONTOCADE* e a sua interface. Como sugestão, a implementação das ferramentas do ambiente pode ser feita a partir do esboço do projeto apresentado na Seção 5.3.1;

e) **Extensão da ferramenta para a fase de implementação de domínio**. Atualmente, a *MADEM* não inclui a fase de implementação de domínio. Esta fase tem como objetivo implementar componentes, geradores de aplicação e linguagens específicas de domínio (LEDs). Em relação às LEDs, sugere-se a extensão do *ONTOCADE* para fornecer suporte à *TOD-DSL* (*Technique based on Ontologies for the Development of Domain Specific Languages*) [59], uma técnica

baseada em ontologias para o desenvolvimento de linguagens específicas de domínio;

f) **Modelagem do *ONTOCADE* usando uma metodologia para Engenharia de Aplicações Multiagentes**. Atualmente, está sendo desenvolvida, no âmbito do projeto *MaAE*, uma metodologia para Engenharia de Aplicações Multiagentes, que pode ser usada na modelagem do ambiente no lugar da *UML*, utilizada neste trabalho.

7 APÊNDICE

Este apêndice apresenta o código-fonte produzido durante a realização deste trabalho para a implementação do *ONTOCADE*. O programa a seguir reúne as classes e métodos específicos do *plugin* e da interface do ambiente.

//ontocade.java

```
package ontocade.br.ufma.deinf.maae; //define o diretório onde serão armazenados os arquivos *.class

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import javax.swing.*;
import javax.swing.table.*;
import java.lang.*;
import java.sql.*;
import edu.stanford.smi.protege.widget.*;
import edu.stanford.smi.protege.action.*;
import edu.stanford.smi.protege.ui.*;
import edu.stanford.smi.protege.event.*;
import edu.stanford.smi.protege.model.*;
import edu.stanford.smi.protege.util.*;
import edu.stanford.smi.protege.resource.*;
import java.awt.Component;
import java.awt.Dimension;
import edu.stanford.smi.protege.util.ComponentFactory.*;

public class ONTOCADE extends AbstractTabWidget{  //classe que leva o nome do arquivo ontocade.java; estende a classe AbstractTabWidget, pertencente à biblioteca do Protégé

  ObjViewPanel painelmodobj;
  PapeisViewPanel painelmodpap;
  IntpapViewPanel painelmodintpap;
  ConceptViewPanel painelmodcon;
  PacpapViewPanel painelmodpacpap;
  FrameViewPanel painelmodframe;
  AgViewPanel painelmodag;
  BehaviorViewPanel painelmodbeh;
  IntagViewPanel painelmodintag;
  PacagViewPanel painelmodpacag;

  DirectInstancesList itsDirectInstancesList;
  ClsesPanel objPanel, papeisPanel, intpapPanel, conPanel;
  JLabel titulo = new JLabel("ONTOLOGY-BASED ENVIRONMENT FOR COMPUTER AIDED DOMAIN ENGINEERING");
  Collection vetor[] = new Collection[100];
  int a;
  private JPanel painel;

  public void initialize() { //método obrigatório que inicia a execução do plugin

    setLabel("ONTOCADE"); // nome que distingue o plugin ONTOCADE dos demais em exibição
    painelmodobj = new ObjViewPanel();
    painelmodpap = new PapeisViewPanel();
    painelmodintpap = new IntpapViewPanel();
    painelmodcon = new ConceptViewPanel();
    painelmodpacpap = new PacpapViewPanel();
```

```
painelmodframe = new FrameViewPanel();
painelmodag = new AgViewPanel();
painelmodbeh = new BehaviorViewPanel();
painelmodintag = new IntagViewPanel();
painelmodpacag = new PacagViewPanel();
```

//componentes rotulados que servem para exibir os modeladores do *ONTOCADE*

```
final LabeledComponent rotmodcon = new LabeledComponent("CONCEPT MODELING", new JScrollPane(painelmodcon));
final LabeledComponent rotmodobj = new LabeledComponent("GOAL MODELING", new JScrollPane(painelmodobj));
final LabeledComponent rotmodpap = new LabeledComponent("ROLE MODELING", new JScrollPane(painelmodpap));
final LabeledComponent rotmodintpap = new LabeledComponent("MODELING OF ROLE INTERACTIONS", new
JScrollPane(painelmodintpap));
final LabeledComponent rotmodpacpap = new LabeledComponent("MODELING OF ROLE PACKAGES", new
JScrollPane(painelmodpacpap));
final LabeledComponent rotmodframe = new LabeledComponent("FRAMEWORK MODELING", new JScrollPane(painelmodframe));
final LabeledComponent rotmodag = new LabeledComponent("AGENT MODELING", new JScrollPane(painelmodag));
final LabeledComponent rotmodbeh = new LabeledComponent("BEHAVIOR MODELING", new JScrollPane(painelmodbeh));
final LabeledComponent rotmodintag = new LabeledComponent("MODELING OF AGENT INTERACTIONS", new
JScrollPane(painelmodintag));
final LabeledComponent rotmodpacag = new LabeledComponent("MODELING OF AGENT PACKAGES", new
JScrollPane(painelmodpacag));


titulo.setFont(new Font( "TimesRoman", Font.BOLD, 18));
```

//botões para alternar entre as atividades da modelagem

```
JButton modconbot = new JButton ("Concept Modeling");
JButton modobjbot = new JButton ("Goal Modeling");
JButton modpapbot = new JButton ("Role Modeling");
JButton modvarbot = new JButton ("Variability Modeling");
JButton modpacbot = new JButton ("Modeling of Role Packages");
JButton modinpbot = new JButton ("Modeling of Role Interactions");
JButton modagebot = new JButton ("Agent Modeling");
JButton modfrabot = new JButton ("Framework Modeling");
JButton modcombot = new JButton ("Behavior Modeling");
JButton modinabot = new JButton ("Modeling of Agent Interactions");
JButton modativbot = new JButton ("Modeling of Agent Packages");
```

//eventos que controlam a exibição do modelador correspondente à atividade de modelagem escolhida

```
modpapbot.addActionListener(
        new ActionListener(){
                public void actionPerformed(ActionEvent e)
                {
                        rotmodobj.setVisible(false);
                        rotmodpap.setVisible(true);
                        rotmodintpap.setVisible(false);
                        rotmodcon.setVisible(false);
                        rotmodpacpap.setVisible(false);
                        rotmodframe.setVisible(false);
                        rotmodag.setVisible(false);
                        rotmodbeh.setVisible(false);
                        rotmodintag.setVisible(false);
                        rotmodpacag.setVisible(false);
                }
        }
        );

modobjbot.addActionListener(
        new ActionListener(){
                public void actionPerformed(ActionEvent e)
                {
                        rotmodobj.setVisible(true);
                        rotmodpap.setVisible(false);
                        rotmodintpap.setVisible(false);
                        rotmodcon.setVisible(false);
                        rotmodpacpap.setVisible(false);
                        rotmodframe.setVisible(false);
                        rotmodag.setVisible(false);
                        rotmodbeh.setVisible(false);
                        rotmodintag.setVisible(false);
                        rotmodpacag.setVisible(false);
```

```
                        }
                }
                );

modinpbot.addActionListener(
                new ActionListener(){
                        public void actionPerformed(ActionEvent e)
                        {
                                rotmodobj.setVisible(false);
                                rotmodpap.setVisible(false);
                                rotmodintpap.setVisible(true);
                                rotmodcon.setVisible(false);
                                rotmodpacpap.setVisible(false);
                                rotmodframe.setVisible(false);
                                rotmodag.setVisible(false);
                                rotmodbeh.setVisible(false);
                                rotmodintag.setVisible(false);
                                rotmodpacag.setVisible(false);

                        }
                }
                );

modconbot.addActionListener(
                new ActionListener(){
                        public void actionPerformed(ActionEvent e)
                        {
                                rotmodobj.setVisible(false);
                                rotmodpap.setVisible(false);
                                rotmodintpap.setVisible(false);
                                rotmodcon.setVisible(true);
                                rotmodpacpap.setVisible(false);
                                rotmodframe.setVisible(false);
                                rotmodag.setVisible(false);
                                rotmodbeh.setVisible(false);
                                rotmodintag.setVisible(false);
                                rotmodpacag.setVisible(false);

                        }
                }
                );

modpacbot.addActionListener(
                new ActionListener(){
                        public void actionPerformed(ActionEvent e)
                        {
                                rotmodobj.setVisible(false);
                                rotmodpap.setVisible(false);
                                rotmodintpap.setVisible(false);
                                rotmodcon.setVisible(false);
                                rotmodpacpap.setVisible(true);
                                rotmodframe.setVisible(false);
                                rotmodag.setVisible(false);
                                rotmodbeh.setVisible(false);
                                rotmodintag.setVisible(false);
                                rotmodpacag.setVisible(false);

                        }
                }
                );

modfrabot.addActionListener(
                new ActionListener(){
                        public void actionPerformed(ActionEvent e)
                        {
                                rotmodobj.setVisible(false);
                                rotmodpap.setVisible(false);
                                rotmodintpap.setVisible(false);
                                rotmodcon.setVisible(false);
                                rotmodpacpap.setVisible(false);
                                rotmodframe.setVisible(true);
                                rotmodag.setVisible(false);
                                rotmodbeh.setVisible(false);
                                rotmodintag.setVisible(false);
```

```
                                        rotmodpacag.setVisible(false);
                        }
        }
        );

modagebot.addActionListener(
        new ActionListener(){
                        public void actionPerformed(ActionEvent e)
                        {
                                        rotmodobj.setVisible(false);
                                        rotmodpap.setVisible(false);
                                        rotmodintpap.setVisible(false);
                                        rotmodcon.setVisible(false);
                                        rotmodpacpap.setVisible(false);
                                        rotmodframe.setVisible(false);
                                        rotmodag.setVisible(true);
                                        rotmodbeh.setVisible(false);
                                        rotmodintag.setVisible(false);
                                        rotmodpacag.setVisible(false);
                        }
        }
        );

modcombot.addActionListener(
        new ActionListener(){
                        public void actionPerformed(ActionEvent e)
                        {
                                        rotmodobj.setVisible(false);
                                        rotmodpap.setVisible(false);
                                        rotmodintpap.setVisible(false);
                                        rotmodcon.setVisible(false);
                                        rotmodpacpap.setVisible(false);
                                        rotmodframe.setVisible(false);
                                        rotmodag.setVisible(false);
                                        rotmodbeh.setVisible(true);
                                        rotmodintag.setVisible(false);
                                        rotmodpacag.setVisible(false);
                        }
        }
        );

modinabot.addActionListener(
        new ActionListener(){
                        public void actionPerformed(ActionEvent e)
                        {
                                        rotmodobj.setVisible(false);
                                        rotmodpap.setVisible(false);
                                        rotmodintpap.setVisible(false);
                                        rotmodcon.setVisible(false);
                                        rotmodpacpap.setVisible(false);
                                        rotmodframe.setVisible(false);
                                        rotmodag.setVisible(false);
                                        rotmodbeh.setVisible(false);
                                        rotmodintag.setVisible(true);
                                        rotmodpacag.setVisible(false);
                        }
        }
        );

modativbot.addActionListener(
        new ActionListener(){
                        public void actionPerformed(ActionEvent e)
                        {
                                        rotmodobj.setVisible(false);
                                        rotmodpap.setVisible(false);
                                        rotmodintpap.setVisible(false);
                                        rotmodcon.setVisible(false);
                                        rotmodpacpap.setVisible(false);
                                        rotmodframe.setVisible(false);
                                        rotmodag.setVisible(false);
                                        rotmodbeh.setVisible(false);
                                        rotmodintag.setVisible(false);
                                        rotmodpacag.setVisible(true);
                        }
```

```
                    }
                    );

        painel = new JPanel();
        painel.add(titulo);

        JLabel texto1 = new JLabel("Domain Modeling");
        JLabel texto2 = new JLabel("Architectural Design");
        JLabel texto3 = new JLabel("Detailed Design");

        texto1.setFont(new Font( "TimesRoman", Font.BOLD + Font.ITALIC, 14));
        texto2.setFont(new Font( "TimesRoman", Font.BOLD + Font.ITALIC, 14));
        texto3.setFont(new Font( "TimesRoman", Font.BOLD + Font.ITALIC, 14));

        painel.add(texto1);
        painel.add(modconbot);
        painel.add(modobjbot);
        painel.add(modpapbot, BorderLayout.CENTER);
        painel.add(modpacbot);
        painel.add(modinpbot);
        painel.add(texto2);
        painel.add(modagebot);
        painel.add(modfrabot);
        painel.add(modativbot);
        painel.add(texto3);
        painel.add(modcombot);
        painel.add(modinabot);
        painel.add(texto4);

        painel.setLayout(new FlowLayout());
        add(painel);
        painel.add(rotmodobj);
        painel.add(rotmodpap);
        painel.add(rotmodintpap);
        painel.add(rotmodcon);
        painel.add(rotmodpacpap);
        painel.add(rotmodframe);
        painel.add(rotmodag);
        painel.add(rotmodbeh);
        painel.add(rotmodintag);
        painel.add(rotmodpacag);

        rotmodobj.setVisible(false);
        rotmodpap.setVisible(false);
        rotmodintpap.setVisible(false);
        rotmodcon.setVisible(false);
        rotmodpacpap.setVisible(false);
        rotmodframe.setVisible(false);
        rotmodag.setVisible(false);
        rotmodbeh.setVisible(false);
        rotmodintag.setVisible(false);
        rotmodpacag.setVisible(false);

        setSize(150, 150);

    }
```

//constrói painel onde é exibido o modelador de objetivos

```
class ObjViewPanel extends JPanel{

        Cls selectedCls = null;


        public ObjViewPanel() {
            setComponents();
            this.add(myCreateDirectInstancesList());

        }

        void setComponents() {
            this.setLayout(new FlowLayout());
        }
```

```
private JSplitPane createClsInstPanel(){

                                        Project project = getProject();
                                        objPanel = new ClsesPanel (project);
                                        JSplitPane splitter = ComponentFactory.createTopBottomSplitPane(true);


                                        FrameRenderer renderer = new FrameRenderer();
                                        renderer.setDisplayDirectInstanceCount(true);
                                        objPanel.setRenderer(renderer);

                                splitter.setTopComponent(objPanel);

                        return splitter;

                                }
```

//cria uma lista de todas as instâncias existentes dos conceitos relacionados à modelagem de objetivos

```
private JComponent myCreateDirectInstancesList(){

        Project project = getProject();

                                        itsDirectInstancesList = new DirectInstancesList(project);
```

//cria uma coleção com todos os *frames* definidos na base de conhecimento

```
                                        Collection classes1 = getProject().getKnowledgeBase().getClses();


                                        for(a = 0; a < 47; a++)
                                          {
                                          vetor[a] = CollectionUtilities.removeFirst(classes1);
                                          classes1 = vetor[a];
                                        }
```

//recupera o item correspondente à modelagem de objetivos

```
                                        Collection classes = vetor[46];
                                        Object o = CollectionUtilities.getFirstItem(classes);
                                        Collection col = CollectionUtilities.createCollection(o);

                                        itsDirectInstancesList.setClses(col);
                                        return itsDirectInstancesList;
                                }

}
```

//mesmo esquema, agora para a modelagem de papéis

```
class PapeisViewPanel extends JPanel{

      Cls selectedCls = null;


      public PapeisViewPanel() {
         setComponents();
         this.add(myCreateDirectInstancesList());

      }

      void setComponents() {
         this.setLayout(new FlowLayout());
      }


private JSplitPane createClsInstPanel(){

                                        Project project = getProject();
                                        papeisPanel = new ClsesPanel (project);
                                        JSplitPane splitter1 = ComponentFactory.createTopBottomSplitPane(true);


                                        FrameRenderer renderer = new FrameRenderer();
```

```
                                        renderer.setDisplayDirectInstanceCount(true);
                                        papeisPanel.setRenderer(renderer);

                                splitter1.setTopComponent(papeisPanel);

                        return splitter1;

                                }

private JComponent myCreateDirectInstancesList(){

        Project project = getProject();

                                        itsDirectInstancesList = new DirectInstancesList(project);

                                        Collection classes1 = getProject().getKnowledgeBase().getClses();

                                        for(a = 0; a < 49; a++)
                                          {
                                           vetor[a] = CollectionUtilities.removeFirst(classes1);
                                           classes1 = vetor[a];
                                        }

                                        Collection classes = vetor[48];
                                        Object o = CollectionUtilities.getFirstItem(classes);
                                        Collection col = CollectionUtilities.createCollection(o);

                                        itsDirectInstancesList.setClses(col);
                                        return itsDirectInstancesList;
                                }


}
```

//mesmo esquema, agora para a modelagem de interação entre papéis

```
class IntpapViewPanel extends JPanel{

      Cls selectedCls = null;


      public IntpapViewPanel() {
         setComponents();
         this.add(myCreateDirectInstancesList());

      }

      void setComponents() {
         this.setLayout(new FlowLayout());
      }


private JSplitPane createClsInstPanel(){

                                        Project project = getProject();
                                        papeisPanel = new ClsesPanel (project);
                                        JSplitPane splitter1 = ComponentFactory.createTopBottomSplitPane(true);


                                        FrameRenderer renderer = new FrameRenderer();
                                        renderer.setDisplayDirectInstanceCount(true);
                                        papeisPanel.setRenderer(renderer);


                                splitter1.setTopComponent(papeisPanel);

                        return splitter1;

                                }

private JComponent myCreateDirectInstancesList(){

        Project project = getProject();

                                        itsDirectInstancesList = new DirectInstancesList(project);
```

```
                                        Collection classes1 = getProject().getKnowledgeBase().getClses();

                                        for(a = 0; a < 54; a++)
                                          {
                                          vetor[a] = CollectionUtilities.removeFirst(classes1);
                                          classes1 = vetor[a];
                                        }

                                        Collection classes = vetor[53];
                                        Object o = CollectionUtilities.getFirstItem(classes);
                                        Collection col = CollectionUtilities.createCollection(o);

                                        itsDirectInstancesList.setClses(col);
                                        return itsDirectInstancesList;
                                }

}
```

//modelagem de conceitos

```
class ConceptViewPanel extends JPanel{

    Cls selectedCls = null;


    public ConceptViewPanel() {
       setComponents();
       this.add(myCreateDirectInstancesList());

    }

    void setComponents() {
       this.setLayout(new FlowLayout());
    }


private JSplitPane createClsInstPanel(){

                                        Project project = getProject();
                                        conPanel = new ClsesPanel (project);
                                        JSplitPane splitter1 = ComponentFactory.createTopBottomSplitPane(true);


                                        FrameRenderer renderer = new FrameRenderer();
                                        renderer.setDisplayDirectInstanceCount(true);
                                        conPanel.setRenderer(renderer);


                                splitter1.setTopComponent(conPanel);

                        return splitter1;

                                }

private JComponent myCreateDirectInstancesList(){

        Project project = getProject();

                                        itsDirectInstancesList = new DirectInstancesList(project);

                                        Collection classes1 = getProject().getKnowledgeBase().getClses();

                                        for(a = 0; a < 56; a++)
                                          {
                                          vetor[a] = CollectionUtilities.removeFirst(classes1);
                                          classes1 = vetor[a];
                                        }

                                        Collection classes = vetor[55];
                                        Object o = CollectionUtilities.getFirstItem(classes);
                                        Collection col = CollectionUtilities.createCollection(o);
```

```
                                        itsDirectInstancesList.setClses(col);
                                        return itsDirectInstancesList;
                                }

}
```

//modelagem de pacotes

```
class PacpapViewPanel extends JPanel{

    Cls selectedCls = null;


    public PacpapViewPanel() {
       setComponents();
       this.add(myCreateDirectInstancesList());

    }

    void setComponents() {
       this.setLayout(new FlowLayout());
    }


private JSplitPane createClsInstPanel(){

                                        Project project = getProject();
                                        conPanel = new ClsesPanel (project);
                                        JSplitPane splitter1 = ComponentFactory.createTopBottomSplitPane(true);


                                        FrameRenderer renderer = new FrameRenderer();
                                        renderer.setDisplayDirectInstanceCount(true);
                                        conPanel.setRenderer(renderer);


                                splitter1.setTopComponent(conPanel);

                        return splitter1;

                                }

private JComponent myCreateDirectInstancesList(){

        Project project = getProject();

                                        itsDirectInstancesList = new DirectInstancesList(project);

                                        Collection classes1 = getProject().getKnowledgeBase().getClses();

                                        for(a = 0; a < 52; a++)
                                          {
                                          vetor[a] = CollectionUtilities.removeFirst(classes1);
                                          classes1 = vetor[a];
                                        }

                                        Collection classes = vetor[51];
                                        Object o = CollectionUtilities.getFirstItem(classes);
                                        Collection col = CollectionUtilities.createCollection(o);


                                        itsDirectInstancesList.setClses(col);
                                        return itsDirectInstancesList;
                                }

}

class FrameViewPanel extends JPanel{

    Cls selectedCls = null;
```

```
    public FrameViewPanel() {
       setComponents();
       this.add(myCreateDirectInstancesList());

    }

    void setComponents() {
       this.setLayout(new FlowLayout());
    }


private JSplitPane createClsInstPanel(){

                                                    Project project = getProject();
                                                    objPanel = new ClsesPanel (project);
                                                    JSplitPane splitter = ComponentFactory.createTopBottomSplitPane(true);


                                                    FrameRenderer renderer = new FrameRenderer();
                                                    renderer.setDisplayDirectInstanceCount(true);
                                                    objPanel.setRenderer(renderer);

                                splitter.setTopComponent(objPanel);

                        return splitter;

                                }

private JComponent myCreateDirectInstancesList(){

        Project project = getProject();

                                                    itsDirectInstancesList = new DirectInstancesList(project);

                                                    Collection classes1 = getProject().getKnowledgeBase().getClses();



                                                    for(a = 0; a < 58; a++)
                                                      {
                                                      vetor[a] = CollectionUtilities.removeFirst(classes1);
                                                      classes1 = vetor[a];
                                                    }

                                                    Collection classes = vetor[57];
                                                    Object o = CollectionUtilities.getFirstItem(classes);
                                                    Collection col = CollectionUtilities.createCollection(o);

                                                    itsDirectInstancesList.setClses(col);
                                                    return itsDirectInstancesList;
                                }

}
```

//modelagem de agentes

```
class AgViewPanel extends JPanel{

    Cls selectedCls = null;


    public AgViewPanel() {
       setComponents();
       this.add(myCreateDirectInstancesList());

    }

    void setComponents() {
       this.setLayout(new FlowLayout());
    }


private JSplitPane createClsInstPanel(){

                                                    Project project = getProject();
```

```
                                        objPanel = new ClsesPanel (project);
                                        JSplitPane splitter = ComponentFactory.createTopBottomSplitPane(true);


                                        FrameRenderer renderer = new FrameRenderer();
                                        renderer.setDisplayDirectInstanceCount(true);
                                        objPanel.setRenderer(renderer);

                                splitter.setTopComponent(objPanel);

                        return splitter;

                                }

private JComponent myCreateDirectInstancesList(){

        Project project = getProject();

                                        itsDirectInstancesList = new DirectInstancesList(project);

                                        Collection classes1 = getProject().getKnowledgeBase().getClses();



                                        for(a = 0; a < 60; a++)
                                          {
                                          vetor[a] = CollectionUtilities.removeFirst(classes1);
                                          classes1 = vetor[a];
                                        }

                                        Collection classes = vetor[59];
                                        Object o = CollectionUtilities.getFirstItem(classes);
                                        Collection col = CollectionUtilities.createCollection(o);

                                        itsDirectInstancesList.setClses(col);
                                        return itsDirectInstancesList;
                                }

}
```

//modelagem de comportamento

```
class BehaviorViewPanel extends JPanel{

      Cls selectedCls = null;


      public BehaviorViewPanel() {
         setComponents();
         this.add(myCreateDirectInstancesList());

      }

      void setComponents() {
         this.setLayout(new FlowLayout());
      }


private JSplitPane createClsInstPanel(){

                                        Project project = getProject();
                                        objPanel = new ClsesPanel (project);
                                        JSplitPane splitter = ComponentFactory.createTopBottomSplitPane(true);


                                        FrameRenderer renderer = new FrameRenderer();
                                        renderer.setDisplayDirectInstanceCount(true);
                                        objPanel.setRenderer(renderer);

                                splitter.setTopComponent(objPanel);

                        return splitter;

                                }
```

```
private JComponent myCreateDirectInstancesList(){

        Project project = getProject();

                                                    itsDirectInstancesList = new DirectInstancesList(project);

                                                    Collection classes1 = getProject().getKnowledgeBase().getClses();


                                                    for(a = 0; a < 65; a++)
                                                      {
                                                       vetor[a] = CollectionUtilities.removeFirst(classes1);
                                                       classes1 = vetor[a];
                                                    }

                                                    Collection classes = vetor[64];
                                                    Object o = CollectionUtilities.getFirstItem(classes);
                                                    Collection col = CollectionUtilities.createCollection(o);

                                                    itsDirectInstancesList.setClses(col);
                                                    return itsDirectInstancesList;
                                        }

}
```

//modelagem de interações entre agentes

```
class IntagViewPanel extends JPanel{

      Cls selectedCls = null;


      public IntagViewPanel() {
         setComponents();
         this.add(myCreateDirectInstancesList());

      }

      void setComponents() {
         this.setLayout(new FlowLayout());
      }


private JSplitPane createClsInstPanel(){

                                                    Project project = getProject();
                                                    objPanel = new ClsesPanel (project);
                                                    JSplitPane splitter = ComponentFactory.createTopBottomSplitPane(true);


                                                    FrameRenderer renderer = new FrameRenderer();
                                                    renderer.setDisplayDirectInstanceCount(true);
                                                    objPanel.setRenderer(renderer);

                                        splitter.setTopComponent(objPanel);

                            return splitter;

                                        }

private JComponent myCreateDirectInstancesList(){

        Project project = getProject();

                                                    itsDirectInstancesList = new DirectInstancesList(project);

                                                    Collection classes1 = getProject().getKnowledgeBase().getClses();


                                                    for(a = 0; a < 67; a++)
                                                      {
                                                       vetor[a] = CollectionUtilities.removeFirst(classes1);
                                                       classes1 = vetor[a];
                                                    }
```

```
                                        Collection classes = vetor[66];
                                        Object o = CollectionUtilities.getFirstItem(classes);
                                        Collection col = CollectionUtilities.createCollection(o);

                                        itsDirectInstancesList.setClses(col);
                                        return itsDirectInstancesList;
                                }

}
```

//procedimento para carregamento do *Protégé* e de suas bibliotecas

```
    public static void main(String[] args) {

        edu.stanford.smi.protege.Application.main(args);
    }
}
```

8 ANEXO

Este anexo mostra a modelagem do ambiente *ONTOCADE* feita com a metodologia *MADEM*. Os diagramas a seguir representam o Modelo de Objetivos, o Modelo de Papéis e o Modelo de Interações.

Modelo de Objetivos

O *ONTOCADE* tem como objetivo geral "Automatizar a aplicação da *MADEM*". Seus objetivos específicos são: "Automatizar a fase de Análise de Domínio da *MADEM*", "Gerenciar as interações do usuário com o *ONTOCADE*" e "Automatizar a fase de Projeto de Domínio da *MADEM*".

O primeiro objetivo específico tem como responsabilidade a realização de cada uma das subtarefas da Análise de Domínio, como por exemplo, a Modelagem de Conceitos e a Modelagem de Interações entre Papéis. Já o terceiro objetivo específico tem as responsabilidades de modelagens específicas à fase de Projeto de Domínio.

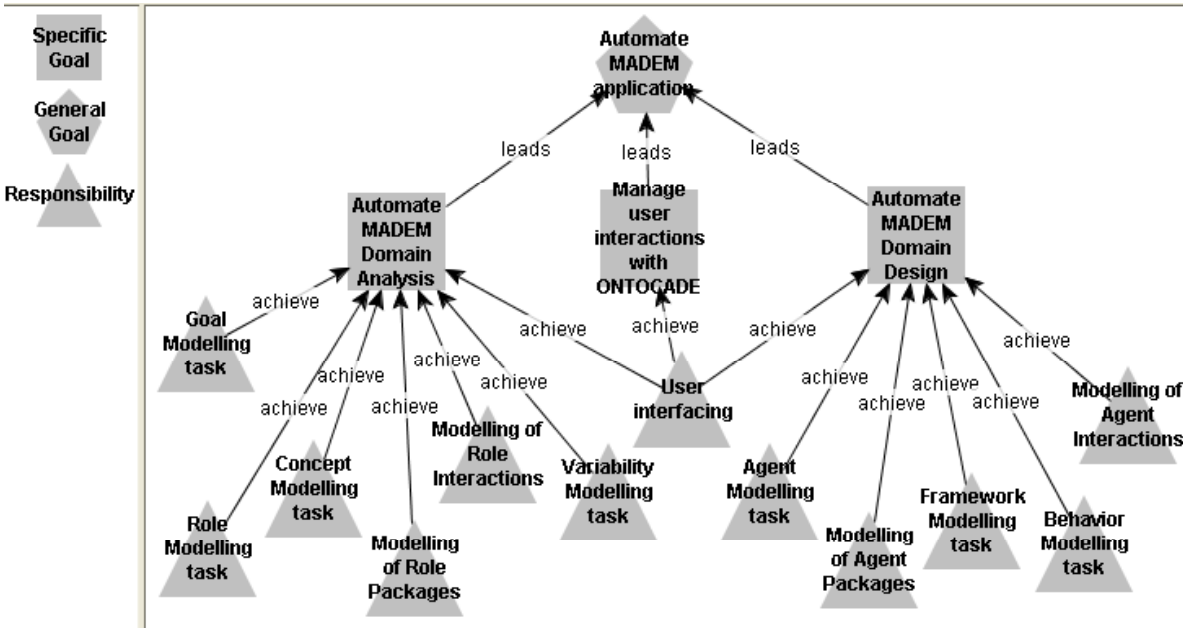

Figura 43 - Modelo de Objetivos do *ONTOCADE*

Modelo de Papéis (Fase de Análise)

A figura a seguir mostra o modelo de papéis relacionados à Análise de Domínio. Cada papel é executado por um modelador que possui uma responsabilidade, sendo esta exercida por uma ou mais atividades. Por exemplo, o Modelador de Objetivos tem a responsabilidade "Tarefa de Modelagem de Objetivos", que é exercida pelas seguintes atividades: "Identificar o objetivo geral", "Identificar objetivos específicos" e "Identificar responsabilidades".

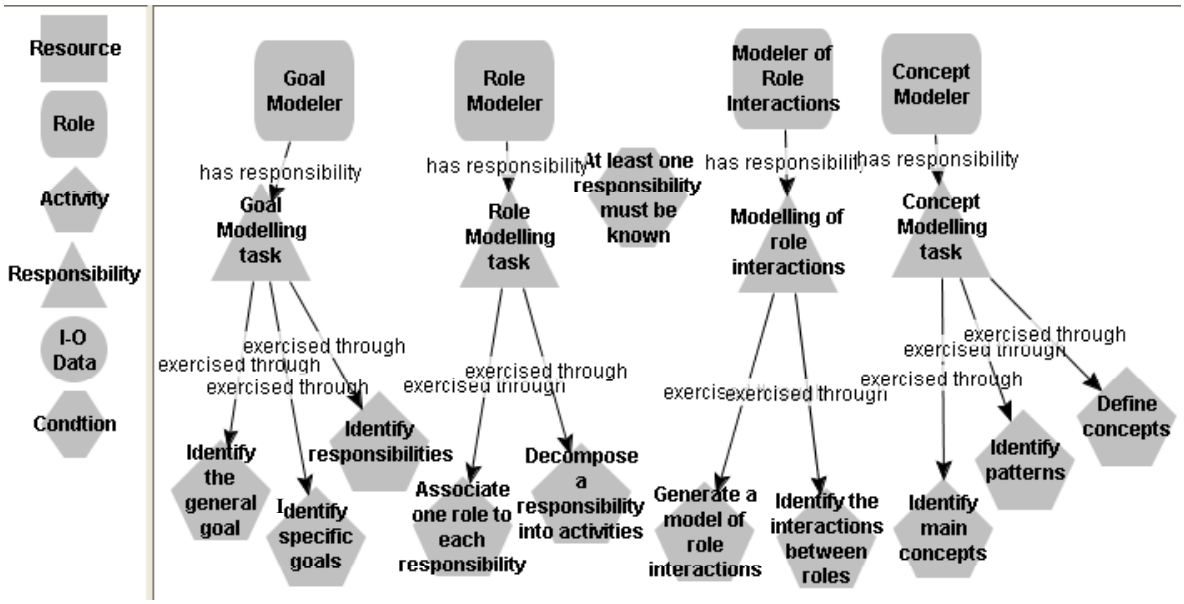

Figura 44 - Modelo de Papéis (fase de Análise de Domínio)

Modelo de Papéis (Fase de Projeto)

Da mesma forma, a figura a seguir mostra o modelo de papéis relacionados ao Projeto de Domínio. Cada papel é executado por um modelador que possui uma responsabilidade, sendo esta exercida por uma ou mais atividades. Por exemplo, o Modelador de Comportamentos possui a responsabilidade "Tarefa de Modelagem de Comportamento", que é exercida pela atividade "Identificar comportamentos a partir das atividades".

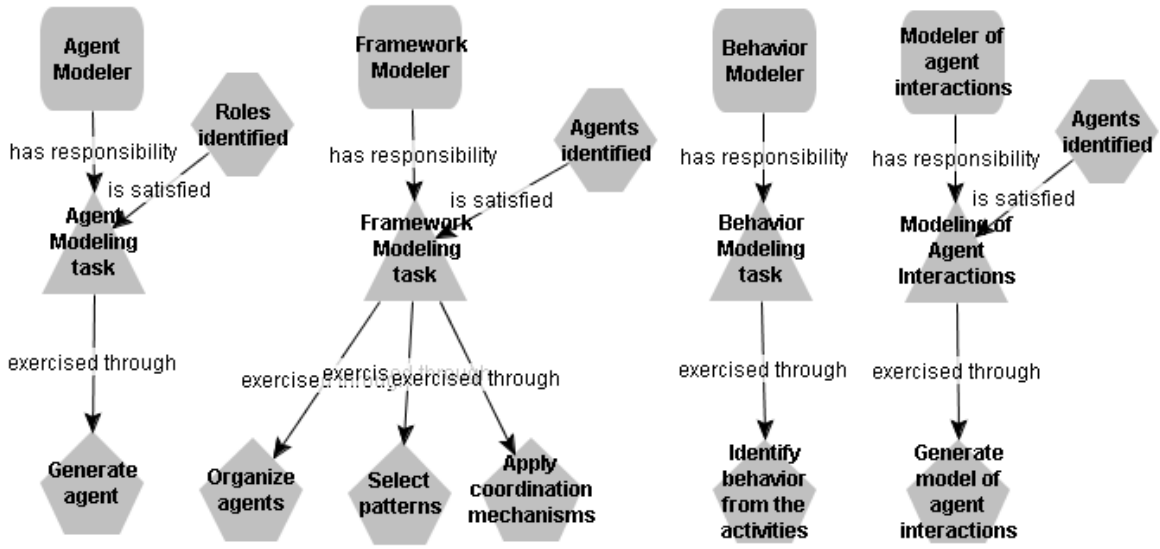

Figura 45 - Modelo de Papéis (fase de Projeto de Domínio)

Modelo de Interações

O Diagrama de Interações mostra como ocorrem as colaborações entre os modeladores do *ONTOCADE*. A numeração indica a ordem em que as interações acontecem.

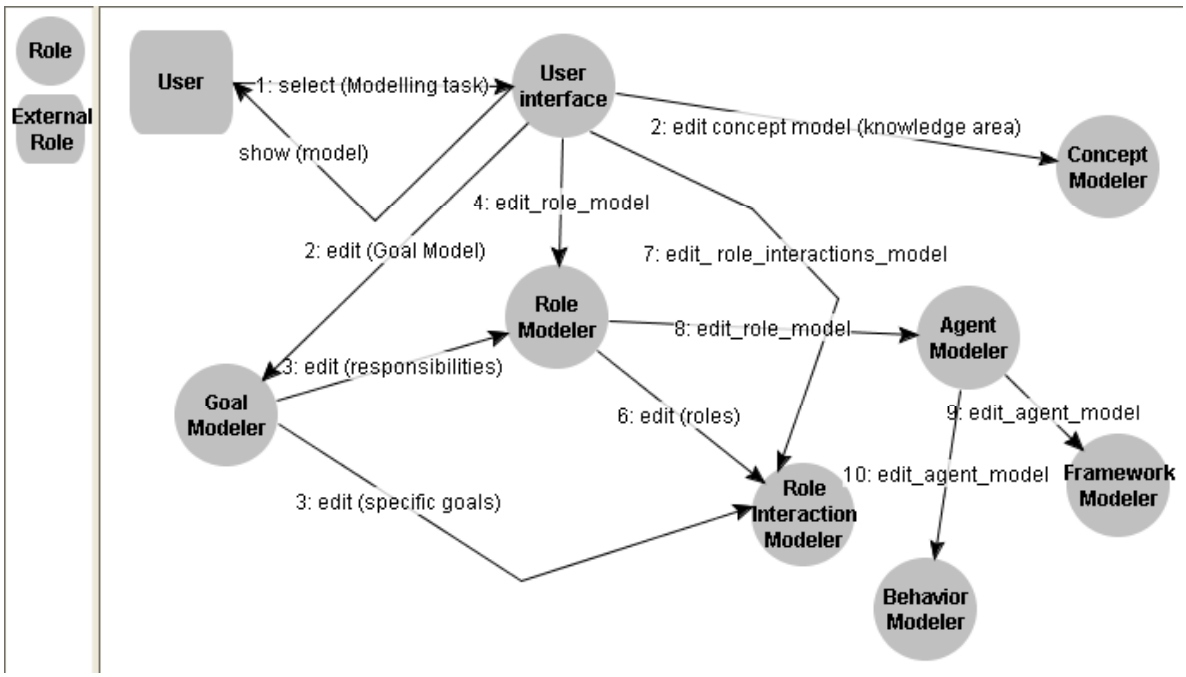

Figura 46 - Modelo de Interações

BIBLIOGRAFIA

[1] BAEZA-YATES, R. & RIBEIRO-NETO, B. **Modern Information Retrieval**. New York: ACM Press Series/Addison Wesley, 1999.

[2] BECHHOFER, S., HORROCKS, I., GOBLE, C. & STEVENS, R. **OilEd: a Reasonable Ontology Editor for the Semantic Web.** In: Working Notes of the 14th International Workshop on Description Logics (DL-2001), p. 1-9, Stanford, EUA, August, 2001.

[3] BOGGS, W., & BOGGS, M. **Mastering UML with Rational Rose 2002**. SYBEX, Inc., 1999.

[4] BOOCH, G., RUMBAUGH, J. & JACOBSON, I. **Unified Modeling Language User Guide**. Reading: Addison Wesley, 1999.

[5] CAIRE, G., LEAL, F., CHAINHO, P., EVANS, R., GARIJO, F., GOMEZ, J., PAVON, J., KEARNEY, P., STARK, J., COULIER, W. & MASSONET, P. **Agent-Oriented Analysis using MESSAGE/UML**. In: Proceedings of the 2nd International Workshop on Agent-Oriented Software Engineering (AOSE 2001), Montreal, Canada, Springer-Verlag, pp. 119-135, 2001.

[6] CASTRO, J., KOLP, M., & MYLOPOULOS, J. **A Requirement-Driven Software Development Methodology**. In: Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAISE 2001), Interlaken, Switzerland, Springer-Verlag, pp. 108-123, 2001.

[7] CHANDRASEKARAN, B. & JOSEHSON, J. **What Are Ontologies, and Why Do We Need Them?** IEEE Intelligent Systems, v. 14, n. 1, pp. 20 – 26, 1999.

[8] CHAUHAN, D. **Developing Coherent Multi-agent Systems using JAFMAS**. In: Proceedings of the International Conference on Multi-agent Systems, ICMAS98, Cite des Sciences – La Villette, Paris, France, July 1998.

[9] CHELLA, A., COSSENTINO, M., & SABATUCCI, L. **Designing JADE Systems with the Support of CASE Tools and Patterns**. In: EXP Online, Vol. 3, n. 3, September, 2003. http://exp.telecomitalialab.com. Acesso em: 06 de dezembro de 2004.

[10] CHELLA, A., COSSENTINO, M., & SABATUCCI, L. **Tools and patterns in designing multi-agent systems with PASSI**. In: Proceedings of the 6th WSEAS International Conference on Telecommunications and Informatics (TELE-INFO 04), Cancun, Mexico, May 2004.

[11] **CLIPS: A Tool for Building Expert Systems**. http://www.ghg.net/clips/CLIPS.html. Acesso em: 06 de dezembro de 2004.

[12] COSSENTINO, M., BURRAFATO, P., LOMBARDO, S., & SABATUCCI, L. **Introducing Pattern Reuse in the Design of Multi-agent Systems**. In: Proceedings of the Agent Technologies, Infrastructures, Tools, and Applications for E-Services (NODe 2002), Agent-Related Workshops, Efurt, Germany, Springer-Verlag, pp. 107-120, 2002.

[13] COSSENTINO, M. & POTTS, C. **A CASE tool supported methodology for the design of multi-agent systems**. In: The 2002 International Conference on Software Engineering Research and Practice (SERP'02) – June 24-27, 2002, Las Vegas (NV), USA.

[14] COSSENTINO, M., SABATUCCI, L., & CHELLA, A. **A Possible Approach to the Development of Robotic Multi-Agent Systems**. IEEE/WIC Conference on Intelligent Agent Technology (IAT'03). October, 13-17, Halifax (Canada), 2003.

[15] DEITEL, H. M., & DEITEL, P. J. **C++: Como Programar**. 3ª edição. Ed. Bookman, 2001.

[16] DEITEL, H. M. & DEITEL, P. J. **Java: Como Programar**. 3ª edição Ed. Bookman, Porto Alegre, 2001.

[17] DELOACH, S. A. **Analysis and Design using MaSE and agentTool**. In: Proceedings of the 12th Midwest Artificial Intelligence and Cognitive Science Conference (MAICS 2001). Miami University, Oxford, Ohio, USA. March 31 – April 1, 2001.

[18] DERIDDER, D. & WOUTERS, B. **The Use of Ontologies as a Backbone for Software Engineering Tools**. In: Fourth Australian Knowledge Acquisition Workshop (AKAW'99), December, 5-6. Sidney (Australia), 1999.

[19] **Extensible Markup Language (XML)**. http://www.w3.org/XML. Acesso em: 06 de dezembro de 2004.

[20] FARIA, C. **Ferramentas para a Engenharia de Domínio e de Aplicações no Desenvolvimento baseado em Agentes**. Relatório Técnico Individual Final PDI-TI-CNPq, Ed. UFMA/DTI-7G-CNPq. São Luís, MA, 2004.

[21] FARIA, C. **Uma Técnica para a Aquisição e Construção de Modelos de Domínio e Usuários baseados em Ontologias para a Engenharia de Domínio Multiagente**, Dissertação (Mestrado em Ciência da Computação) – Curso de Pós-Graduação em Engenharia de Eletricidade, Universidade Federal do Maranhão – UFMA, 2004.

[22] FARQUHAR, A., FIKES, R. & RICE, J. **The Ontolingua Server: a Tool for Collaborative Ontology Construction**. Int. J. Human-Computer Studies, 46, p. 707-727, Knowledge Systems Laboratory, Stanford University. Stanford, CA, USA, 1997.

[23] FERREIRA, S. **Uma Técnica e uma Ferramenta para o Projeto de Domínio Global e Detalhado de Sistemas Multiagentes**, Dissertação (Mestrado em Ciência da Computação) – Curso de Pós-Graduação em Engenharia de Eletricidade, Universidade Federal do Maranhão – UFMA, 2004.

[24] **FIPA**. http://www.fipa.org. Acesso em: 06 de dezembro de 2004.

[25] **FIPA ACL**. http://www.fipa.org/specs/fipa00061. Acesso em: 06 de dezembro de 2004.

[26] FOREMAN, J. **Product Line based Software Development – Significant Results, Future Challenges**. Software Technology Conference, Salt Lake City, UT, April 23, 1996.

[27] FRAWLEY, W.J., PIATETSKY-SHAPIRO, G. & MATHEUS, C.J. **Knowledge discovery in databases: an overview**. In: "Knowledge Discovery in Databases", G. Piatetsky-Shapiro & W.J. Frawley, editors. AAAI / MIT Press, 1991.

[28] GIRARDI, R. **Engenharia de *Software* baseada em Agentes**. Anais do IV Congresso Brasileiro de Ciência da Computação (CBCOMP 2004), Ed. UNIVALI, pp. 913-937. Itajaí, Santa Catarina, Brasil, 08 a 12 de outubro, 2004.

[29] GIRARDI, R. **Reuse in Agent-based Application Development**. In: Proceedings of the 1st International Workshop on *Software* Engineering for Large-Scale Multi-Agent Systems (SELMAS'2002), Orlando, Florida (USA), May 19, 2002.

[30] GIRARDI, R., FARIA, C. & BALBY, L. **Ontology-based Domain Modeling of Multi-Agent Systems**. In: Proceedings of the Third International Workshop on Agent-

Oriented Methodologies at International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2004). 24 a 28 de outubro de 2004.

[31] GIRARDI, R, & SODRÉ, A. C. S. **A Methodology for Multi-Agent Application Development**. In: 6th International Conference on Intelligent Tutoring Systems, Proceedings of the ITS´2002 Workshops - Architectures and Methodologies for Building Agent-Based Learning Environments, Biarritz, v. 1, pp. 58-66, 2002.

[32] HARRISON, W., OSSHER, H. & TARR, P. **Software Engineering Tools and Environments: A Roadmap**. In: Proceedings of the Conference on the Future of Software Engineering, pp. 261-277, Limerick (Ireland), 2000.

[33] **JADE - Java Agent Development Framework**. http://jade.tilab.com. Acesso em: 06 de dezembro de 2004.

[34] **Java Technology**. http://java.sun.com. Acesso em: 06 de dezembro de 2004.

[35] JOHNSON, R. E. **Frameworks = (Components + Patterns): How frameworks compare to other object-oriented reuse techniques**. Communications of the ACM, v. 40, n. 10, 1997.

[36] KNUBLAUCH, H., FERGERSON, R. W., NOY, N. F. & MUSEN, M. A. **The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications**. In: Proceedings of the Third International Semantic Web Conference (ISWC'04), November 7-11, Hiroshima (Japan), 2004.

[37] MAHALINGAM, K. & HUHNS, M. N. **A Tool for Organizing Web Information**. In: IEEE Computer, v. 30, n. 6, p. 80-83, June, 1997.

[38] MARTIN, J. **Information Engineering: introduction**, Prentice Hall, 178 p., 1991.

[39] **MAST – Multi-Agent Systems Tool**. http://www.gsi.dit.upm.es/~mast. Acesso em: 06 de dezembro de 2004.

[40] MENKEN, M. **JESS Tutorial**. http://www.cs.vu.nl/~ksprac/export/jess-tutorial.pdf. Acesso em: 06 de dezembro de 2004.

[41] MICHAHELLES, F. **CASE tools as toolkits for suppliers**, Course Paper – Massachusetts Institute of Technology – Sloan School of Management, 2000.

[42] **Multi-Agents Systems**. http://www.multiagent.com. Acesso em: 06 de dezembro de 2004.

[43] NUSEIBEH, B. **Meta-CASE Support for Method-Based Software Development**. In: Proceedings of the 1st International Congress on Meta-*CASE*, Sunderland, UK, January, 1995.

[44] ODELL, J., PARUNAK, H. D. & BAUER, B. **Extending UML for agents**. In: Proceedings of the 2nd International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS 2000), Austin, USA, pp. 3–17, 2000.

[45] **OIL - Ontology Inference Layer**. http://www.ontoknowledge.org/oil. Acesso em: 06 de dezembro de 2004.

[46] **OILEd**. http://oiled.man.ac.uk. Acesso em: 06 de dezembro de 2004.

[47] OLIVEIRA DE ALMEIDA, H. **COMPOR: Desenvolvimento de Software para Sistemas Multiagentes**. Dissertação (Mestrado em Ciência da Computação), Departamento de Ciência da Computação, Universidade Federal de Campina Grande – UFCG, 2004.

[48] OLIVEIRA, I. **Um Sistema de Padrões baseados em Agentes para a Modelagem de Usuários e Adaptação de Sistemas**, Dissertação (Mestrado em Ciência da Computação) – Curso de Pós-Graduação em Engenharia de Eletricidade, Universidade Federal do Maranhão – UFMA, 2004.

[49] OLIVEIRA, I., SOUSA, R. & GIRARDI, R. **Uma Ontologia para a Especificação de Sistemas de Padrões**. Anais da Quarta Conferência Latino-Americana em Linguagens de Padrões para Programação – SugarLoafPLOP2004, 10-13 agosto, Fortaleza-CE, 2004.

[50] OMICINI, A. **SODA Societies and Infrastructures in the Analysis and Design of Agent-based Systems**. In: Proceedings of the First International Workshop, AOSE 2000 on Agent-Oriented Software Engineering, Limerick, Ireland, pp. 185-193, January, 2001.

[51] **OntoEdit**. http://www.ontoknowledge.org/tools/ontoedit.shtml. Acesso em: 06 de dezembro de 2004.

[52] **OWL - Web Ontology Language**. http://www.w3.org/TR/owl-features. Acesso em: 06 de dezembro de 2004.

[53] PRESSMAN, R. S. **Software Engineering**, 5th Edition, McGraw-Hill Interamericana, 2002.

[54] PRIETO-DÍAZ, R. **Domain Analysis and Software Systems Modeling**. Los Alamitos, CA: IEEE Computer Society Press, 312 p., 1991.

[55] **Protégé Project**. http://protege.stanford.edu. Acesso em: 06 de dezembro de 2004.

[56] **Resource Description Framework (RDF)**. http://www.w3.org/RDF. Acesso em: 06 de dezembro de 2004.

[57] SALTON, G. & MCGILL, M. **An Introduction to Modern Information Retrieval**. New York: McGraw-Hill. 1983.

[58] SANDHOLM, T. & LESSER, V. **Issues in Automated Negotiation and Electronic Commerce: Extending the Contract Net Framework**. Reading in Agents, M. Huhns and M. Singh (eds.), Morgan Kaufmann Publishers, pp. 66-73. January 1997.

[59] SERRA, I. **Uma Técnica para o Desenvolvimento de Linguagens Específicas de Domínio**, Dissertação (Mestrado em Ciência da Computação) – Curso de Pós-Graduação em Engenharia de Eletricidade, Universidade Federal do Maranhão – UFMA, 2004.

[60] SERRA, I., GIRARDI, R. & SILVA FILHO, J. H. A. **Um Modelo de Domínio baseado em Ontologias para a Recuperação e Filtragem de Informação**. Anais do IV Congresso Brasileiro de Ciência da Computação (CBCOMP 2004), Ed. UNIVALI, pp. 124-129. Itajaí, Santa Catarina, Brasil. 08 a 12 de outubro de 2004.

[61] SILBERSCHATZ, A., KORTH, H. F. & SUDARSHAN, S. **Sistema de Banco de Dados**. MAKRON Books, 3ª ed., 1999.

[62] SILVA FILHO, J. H. A. **SIMCAP: Um Sistema Multiagente para a Captura de Publicações Científicas na Web.** Monografia de Graduação – Curso de Ciência da Computação, Universidade Federal do Maranhão – UFMA, 2001.

[63] SILVA JUNIOR, G. B. **Padrões Arquiteturais para o Desenvolvimento de Aplicações Multiagentes**, Dissertação (Mestrado em Ciência da Computação) – Curso de Pós-Graduação em Engenharia de Eletricidade, Universidade Federal do Maranhão – UFMA, 2003.

[64] SOMMERVILLE, I. **Software Engineering**. 6th Edition. Addisson-Wesley Publishing Company. Lancaster University, 2000.

[65] STAAB, S. & MAEDCHE, A. **Ontology Engineering beyond the Modeling of Concepts and Relations**. In: Proceedings of the European Conference on Artificial Intelligence, Workshop on Applications of Ontologies and Problem-Solving Methods, IOS Press, Amsterdam, 2000.

[66] SURE, Y., ERDMANN, M., ANGELE, J., STAAB, S., STUDER, R. & WENKE, D. **OntoEdit: Collaborative Ontology development for the Semantic Web**. In: Proceedings of the First International Semantic Web Conference 2002 (ISWC 2002), June 9-12, Sardinia, Italia, 2002.

[67] SWICK, R., MILLER, E., SCHLOSS, B., SINGER, D. & BRICKLEY, D. **Resource Description Framework (RDF).** W3C, http://www.w3c.org/RDF, 2000.

[68] TESSEM, B., BJORNESTAD, S., TORNES, K. M. & STEINE-ERIKSEN, G. **ROSA = Reuse of Object-oriented Specifications through Analogy: A Project Framework**. Report nº 16 – Department of Information Science, University of Bergen, Norway. March, 1994.

[69] USCHOLD, M. & GRUNINGER, M. **Ontologies: Principles, Methods and Applications**. Knowledge Engineering Review. Vol. 11 - Number 2. February, 1996.

[70] WOOD, M., DELOACH, S., & SPARKMAN, C. H. **Multi-Agent System Engineering,** The International Journal of Software Engineering and Knowledge Engineering. v. 11(3), June 2001.

[71] WOOLDRIDGE, M., JENNINGS, N. & KINNY, D. **The Gaia Methodology for Agent-Oriented Analysis and Design,** International Journal of Autonomous Agents and Multi-Agent Systems, v. 3, 285 - 312, 2000.

Silva Filho, José Henrique Alves da

ONTOCADE: Um Ambiente *CASE* baseado em Ontologias para Análise e Projeto na Engenharia de Domínio Multiagente / José Henrique Alves da Silva Filho. – São Luís, 2005.

129 f.: il.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal do Maranhão, 2005.

1. Sistemas Multiagentes. 2. Ambientes *CASE*. 3. Engenharia de Domínio. I. Título.

CDU 004.891